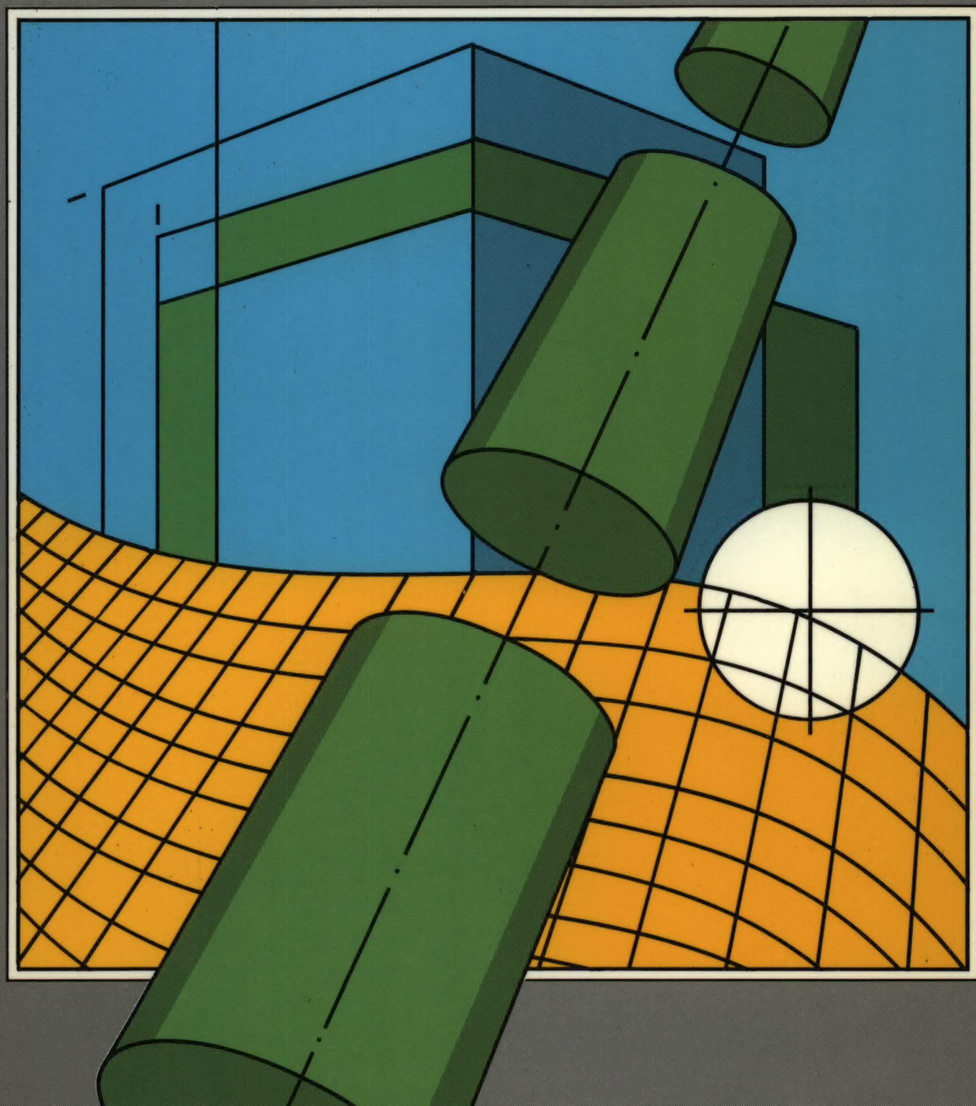
Een inleiding tot gegevenstypen

J. Craig Cleaveland

ACADEMIC SERVICE ADDISON-WESLEY



Een inleiding tot gegevenstypen

Serie onder redactie van ir. J.J. van Amstel:

Compilerbouw - Wirth
Database, een inleiding - Date
Een inleiding tot gegevenstypen - Cleaveland
Inleiding in de berekenbaarheidstheorie - Rayward-Smith
Inleiding in de theorie van formele talen - Rayward-Smith
Inleiding systeemanalyse en systeemontwerp - Davis
Inleiding in de formele logica met toepassingen in de informatica - Dowsing, Rayward-Smith & Walter (in voorbereiding)
Problemen oplossen met de computer - Dromey
Theorie en praktijk van besturingssystemen - Peterson & Silberschatz

Serie in samenwerking met Addison-Wesley:

Basiskennis bestuurlijke informatiekunde - O'Leary & Williams
Bestuurlijke informatieverzorging - van Swigchem
Database, een inleiding - Date
Gegevensbanken, een inleiding - Blanken & Date

Een inleiding tot gegevenstypen

J. Craig Cleaveland

ACADEMIC SERVICE ADDISON-WESLEY

Oorspronkelijke titel: *An Introduction to Data Types* uitgegeven door Addison-Wesley Publishing Company, Inc.

Vertaling: L. Geurts

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Cleaveland, J. Craig

Een inleiding tot gegevenstypen / J. Craig Cleaveland ;
[vert. uit het Engels door L. Geurts]. - Schoonhoven : Academic Service
Vert. van: An introduction to data types. - Addison-Wesley, 1986. - Met lit. opg.
ISBN 90-6233-292-7
SISO 529 SVS 8.12.3 UDC 681.3.01 NUGI 852
Trefw.: gegevenstypen.

10 9 8 7 6 5 4 3 2 1

Uitgegeven door: Academic Service
Postbus 81
2870 AB Schoonhoven
Zetwerk: Redactiebureau R. Heyer, Markelo
Omslag: BSE Advertising, Alphen a/d Rijn
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam

ISBN 90 6233 292 7
NUGI 852

Inhoud

Voorwoord

Terwijl ik dit boek aan het schrijven was, werd me wel eens gevraagd wat het verschil is tussen gegevenstypen en gegevensstructuren. Ik legde het verschil uit aan de hand van een vergelijking:

> gegevenstypen verhouden zich tot gegevensstructuren
> als
> programmeertalen tot programma's.

Gegevenstypen zijn speciale constructies in programmeertalen die worden gebruikt om gegevensstructuren te beschrijven en te definiëren. Deze uitleg helpt wel wat, maar toch zullen sommigen nog hun hoofd schudden. In dit boek worden programmeertalen besproken: ontwerp, implementatie en specificatie. Het behandelt geen zaken die betrekking hebben op gegevensstructuren, algoritmen of efficiëntie, hoewel er wel enkele nieuwe gegevensstructuren worden besproken, zoals procedurele gegevensstructuren. Hoewel er vele boeken over gegevensstructuren en vele boeken over programmeertalen bestaan, is dit één van de eerste boeken die uitsluitend aan gegevenstypen is gewijd.

Omdat gegevenstypen voorzieningen in programmeertalen zijn, kan men dit boek met een gerust hart classificeren als een gespecialiseerd boek over programmeertalen. Het kan samen met andere boeken worden gebruikt voor colleges over de principes van programmeertalen, over het ontwerp van programmeertalen, over gegevensstructuren of over de semantiek van programmeertalen. Het kan als uitgangspunt dienen voor inleidende colleges over gegevenstypen. Het boek is ook nuttig als naslagwerk op het gebied van programmeertalen.

Hoofdstuk 1 behoort vóór de andere hoofdstukken te worden gelezen, omdat het de basis vormt voor datgene wat daarna wordt behandeld. De rest van het boek is opgebouwd uit vier delen die elk uit drie hoofdstukken bestaan:

I. Een overzicht van gegevenstypen
II. Kwesties rond gegevenstypen
III. Gegevensabstracties
IV. Specificaties

De hoofdstukken van de laatste twee delen (hoofdstuk 8-10 en 11-13) kunnen waarschijnlijk het beste bestudeerd worden in de gegeven volgorde. De overige hoofdstukken kunnen in elke gewenste volgorde worden bestudeerd zonder dat de stof onbegrijpelijk wordt. Deel II (hoofdstuk 2-4) bevat veel stof over specifieke programmeertalen en typensystemen en weinig theorie. Het is meer bedoeld om de evolutie van gegevenstypen zoals ze in vele talen worden aangetroffen in historisch perspectief te plaatsen. Daarom moet daar af en toe gebruik gemaakt worden van begrippen die pas in latere hoofdstukken in detail worden beschreven. Deze stof is weliswaar niet nodig om de latere hoofdstukken te kunnen begrijpen, maar plaatst alles wel in een historisch en praktisch kader.

Vele programmeertalen hebben bijgedragen aan de talloze ideeën in dit boek. Vele ideeën en voorbeelden die in dit boek worden gebruikt, stammen uit een groot aantal bronnen, waaronder de volgende:

Ada	Habermann en Perry (1983), Ichbiah et al. (1979)
ALGOL 60	Naur (1963)
ALGOL 68	Van Wijngaarden et al. (1975), Tanenbaum (1976)
Alphard	Wulf et al. (1976)
APL	Falkoff en Iverson (1973, 1978)
BASIC	Harle (1983)
BCPL	Richards (1969)
Bliss	Wulf et al. (1971)
C	Kernighan en Ritchie (1978)
CLU	Liskov et al. (1977)
COBOL	Jackson (1977)
EL1	Wegbreit (1974)
Euclid	Lampson et al. (1977), Popek et al. (1977)
FORTRAN	Brainerd (1978)
FP	Backus (1978)
HOPE	Burstall et al. (1980)
ICON	Griswold (1982, 1983), Wampler en Griswold (1983)
LISP	McCarthy (1960), McCarthy en Levin (1965), Allen (1978)
Mesa	Geschke et al. (1977)

ML	Gordon et al. (1979)
Modula	Wirth (1977)
Modula-2	Wirth (1980)
Pascal	Wirth (1971), Hoare en Wirth (1973), Jensen en Wirth (1974), Welsh et al. (1977), Addyman (1980)
PL/I	Beech (1970)
Russell	Demers et al. (1978, 1980)
SETL	Schonberg et al. (1981)
SIMULA	Dahl et al. (1968)
Smalltalk	Goldberg et al. (1983)
SNOBOL	Griswold et al. (1971)

Verschillende delen van dit boek zijn getest in cursussen, zowel in de academische als in de industriële wereld. Ik dank de studenten die mij hebben geholpen de stof in cursussen te testen, met name Chuck Bullis, John Sherman en Tom Wetmore. Ik dank ook degenen die het manuscript hebben gelezen en hun vele nuttige aanbevelingen zijn in dit boek opgenomen: Paul Eggert en Samuel Kamin van de Universiteit van Illinois, Henry Ledgard en Nancy Leveson van de Universiteit van Californië in Irvine, en Jon Shultis van de Universiteit van Colorado in Boulder. En ik dank mijn vrouw Tina, die er vele avonden aan heeft besteed om ons boek op kleine en grote punten te verbeteren.

North Andover, Mass.

JCC

1 Inleiding

1.1 Wat is een type?

Een van de eigenaardigheden van de mens is de behoefte dingen in te delen in verschillende categorieën, die we typen zullen noemen. In veel gevallen is een type gewoon een verzameling dingen; een systeem van typen, of kortweg een typensysteem, is een manier om verschillende typen te ordenen. We ontwikkelen zulke systemen om ingewikkelde verbanden gemakkelijker te kunnen doorgronden. Zo maken programmeertalen gebruik van gegevenstypen om gegevens te classificeren en bepaalde soorten fouten te voorkomen. We kunnen inzicht krijgen in de manier waarop programmeertalen typen gebruiken door eerst typensystemen buiten de informatica te bekijken.

Woorden zijn het bekendste voorbeeld van typen. We gebruiken bijvoorbeeld zelfstandige naamwoorden om de voorwerpen in onze ingewikkelde wereld in te delen in kleinere, meer begrijpelijke verzamelingen van dingen. De meeste zelfstandige naamwoorden definiëren een of ander type voorwerp. Zo definieert het woord *tafel* de verzameling van alle voorwerpen die we als tafel beschouwen. Evenzo definiëren werkwoorden, bijvoeglijke naamwoorden en andere woordsoorten klassen van respectievelijk handelingen, eigenschappen en andere soorten benoembare zaken.

De taxonomie is een classificatiesysteem in de biologie. Het is een hiërarchische classificatie met een aantal niveaus (van regnum of rijk op het hoogste niveau tot soorten op het laagste niveau). Biologen classificeren elk organisme als lid van een of andere soort, elke soort als lid van een of ander geslacht, enzovoort. Anders dan bij woorden uit het alledaagse taalgebruik, waar vele ver-

schillende woorden hetzelfde voorwerp kunnen aanduiden, behoort ieder organisme tot slechts één soort.

Een gebruikelijke manier om boeken in te delen is naar onderwerp. Een algemeen onderwerp wordt daarbij verdeeld in kleinere deelgebieden, die weer verder kunnen worden onderverdeeld. Hier zien we ook een hiërarchische structuur. We kunnen bijvoorbeeld de volgende indeling maken: non-fiction, exacte wetenschappen, natuurkunde, quantumfysica, quantumchromodynamica. Anders dan in de biologische taxonomie kan in dit systeem iets ook in een algemene klasse worden ingedeeld in plaats van in één bepaalde klasse. Een dier zal altijd tot een bepaalde soort behoren, maar een inleidend natuurkundeboek kan waarschijnlijk niet goed bij een bepaald gebied uit de natuurkunde worden ingedeeld. Bepaalde boeken kunnen bijzonder problematisch zijn. Moet een boek over de geschiedenis van de wiskunde onder geschiedenis of onder wiskunde worden ingedeeld? Moeten boeken over computers onder wiskunde worden geclassificeerd of onder technische wetenschappen, of misschien onder een nieuw onderwerp informatica? Het kan ook zeer frustrerend zijn multidisciplinaire boeken, die verscheidene gebieden bestrijken, onder te brengen in een star classificatiesysteem.

De classificatie van natuurkundige grootheden is eenvoudig vergeleken bij het classificeren van boeken. Elke grootheid is gekoppeld aan precies één dimensie, zoals tijd, oppervlakte, energie, dichtheid of druk. Dat is geen hiërarchie, maar wel een elegante algebraïsche structuur, een zogenaamde *Abelse groep*.

Wiskundigen classificeren abstracte objecten in verschillende typen, waaronder verzamelingen, functies, getallen en relaties. De relaties tussen deze verschillende typen hangen echter af van wat de wiskundige met de typen wil doen. Een verzamelingstheoreticus kan alles weergeven met behulp van verzamelingen. Maar functies, relaties en zelfs getallen kunnen zelf worden gebruikt om verzamelingen weer te geven. We zouden dus een willekeurige hiërarchie uit deze typen kunnen construeren. Ondanks al deze gezichtspunten gaat het om duidelijk te onderscheiden typen. Ieder object behoort tot precies één type; maar objecten van het ene type kunnen objecten van andere typen voorstellen.

Laten we tenslotte eens kijken naar de numerieke typen in FORTRAN, zoals INTEGER, REAL en DOUBLE PRECISION. Elk van deze typen specificeert een verzameling getallen. Een in een FORTRAN-programma gebruikte getalswaarde kan maar tot één van deze typen behoren. Deze beperking roept een probleem op dat lijkt op dat bij het classificeren van boeken. Tot welk numeriek

type behoort getal twee? Om deze vraag te kunnen beantwoorden, moeten we eerst beseffen dat de waarden die tot een numeriek type behoren, geen getallen zijn. De waarden die tot een numeriek type behoren, *stellen* getallen *voor*. Elk numeriek type heeft zijn eigen manier om een getal te representeren. Zo is er een INTEGER twee, een REAL twee en een DOUBLE PRECISION twee. We kunnen het verschil ertussen syntactisch vaststellen. De string ‘2’ stelt bijvoorbeeld de integer-waarde voor en de string ‘2.0’ stelt de floating-point-waarde voor.

1.2 Argumenten voor het gebruik van typen

De drie belangrijkste argumenten om typen te gebruiken zijn:

1. Typen helpen ons onze ideeën over objecten te begrijpen en er systeem in aan te brengen.
2. Een typensysteem helpt ons de unieke eigenschappen van bepaalde typen te zien en erover te discussiëren.
3. Typen helpen ons fouten te ontdekken.

Niet al deze argumenten zijn bepalend voor het opzetten van een typensysteem, maar in programmeertalen zijn ze alle drie van belang. Laten we ze eens nader onder de loep nemen.

Allereerst stellen typen de programmeur in staat een oplossing voor een probleem te bedenken en te formuleren door bepaalde kenmerken van gegevens een naam te geven en te definiëren. Zo is het gebruik van typen een belangrijke schakel tussen de echte wereld en de gegevens waar het programma mee werkt. Een typensysteem stelt ons in staat onze aandacht te beperken tot een bepaald soort object. Zo geeft de uitspraak

'*x* is van het type *y*'

informatie over *x* (en soms over *y*). Nog enkele voorbeelden zijn:

1. Dat is een stoel.
2. Joost van den Vondel is een *Homo sapiens*.
3. *Gijsbrecht van Aemstel* is een toneelstuk.
4. Een meter is een lengtemaat.
5. Laat *V* een verzameling zijn.
6. INTEGER X

Voorbeeld 6 is in FORTRAN geschreven en betekent: 'laat X een integer variabele zijn'. Merk op dat elk van de bovenstaande uitspraken wel iets zegt over het type van een object, maar niets over de waarde of de inhoud ervan. De uitspraken classificeren de objecten alleen. Een heel ander soort uitspraak is: '*x* is *y*'. Hierdoor wordt aangegeven dat *x* hetzelfde ding is als *y*. Hier zijn zes voorbeelden:

1. Dat is de stoel.
2. Joost van den Vondel is degene die *Gijsbrecht van Aemstel* heeft geschreven.
3. *Gijsbrecht van Aemstel* is het toneelstuk dat door Joost van den Vondel is geschreven.
4. Een meter is 100 cm.
5. Laat *V* de verzameling van alle even getallen zijn.
6. X .EQ. 5

Voorbeeld 6 is eveneens in FORTRAN geschreven en vraagt: 'is de waarde van X gelijk aan 5?'

Op de tweede plaats zeggen typen ons iets over de eigenschappen van de objecten waar we mee werken, want elk type heeft zijn eigen eigenschappen. 'Aantal poten' behoort bij tafels en dieren, maar niet bij huizen en planten. Relaties kunnen reflexief zijn, maar 'reflexief getal' heeft geen gangbare wiskundige betekenis. Lengte hoort bij arrays en strings, maar niet bij logische waarden. Soms zijn de eigenschappen van een type zo belangrijk, dat ze het type geheel bepalen. Die situatie komt in de biologie en in de wiskunde vaak voor. Zoogdieren hebben haar, zijn warmbloedig en geven hun jongen melk. Een equivalentierelatie is een relatie die reflexief, symmetrisch en transitief is.

Tenslotte vormt het kunnen ontdekken van fouten een krachtig argument om typen te gebruiken. Een typensysteem kan ons in staat stellen het onjuiste gebruik van een object te ontdekken. Als een object van type *x* wordt gebruikt in een context die niet past bij een object van een type *x*, is er sprake van een *typefout*. De eigenschappen van een type leveren aanwijzingen voor het opsporen van typefouten. De volgende uitspraken bevatten typefouten:

1. Die tafel heeft twee deuren en vier ramen.
2. Joost van den Vondel heeft vier takken.
3. Die biografie vormt een geromantiseerde verhandeling over quarks.
4. (4 meter / 6 seconden) * 3 seconden = 2 seconden

5. $(A \subset B) \cup A$
6. X = 3 + "DAG"

Typecontrole is één van de meest effectieve manieren waarop een compiler fouten kan ontdekken.

Een typensysteem is dus een methode om een verzameling objecten in te delen in verschillende categorieën, die we typen noemen. We gebruiken vaak unieke eigenschappen om een type te definiëren. We gebruiken typensystemen om ideeën te begrijpen en over te brengen en om fouten te ontdekken. We hebben aandacht geschonken aan typensystemen in het algemeen en aan enkele voorbeelden van typensystemen. We hebben gezien dat typensystemen kunnen worden gedefinieerd door hun algebraïsche eigenschappen of door hun hiërarchische structuur. De classificatie van typensystemen is op zichzelf dus weer een typensysteem. In dit boek gaan we in op de typensystemen van programmeertalen.

1.3 Wat is een gegevenstype?

Een type is een verzameling waarden. Dat was de aanvankelijke opvatting over gegevenstypen, die wegens zijn eenvoud en elegantie nog steeds in zwang is. Deze definitie legt de vinger op het wezen van elk classificatiesysteem. Om een type te definiëren behoef je alleen een verzameling waarden aan te geven. Dat kan gebeuren door middel van opsomming of met behulp van eigenschappen. Als werkhypothese kunnen we met deze definitie een heel eind komen, maar juist omdat deze opvatting zo eenvoudig is, schiet die voor het hedendaagse begrip gegevenstype te kort.

Types are not sets (typen zijn geen verzamelingen). Dit was de titel van een artikel van Jim Morris (1973). Het was een bondige weergave van de voortdurende discussie over gegevenstypen. Onze opvatting over de aard van gegevenstypen is in feite sinds de opkomst van programmeertalen in ontwikkeling geweest en zal blijven veranderen en groeien. Dat proces leidt vaak tot heftige discussies die zich ook uitstrekken tot een aantal verwante kwesties en problemen. Veel van die problemen hebben betrekking op de aard van waarden, variabelen, symbolen en objecten.

In het begin van de jaren zeventig begon een revolutie: de introductie van abstracte gegevenstypen. Het idee van abstracte gegevenstypen vormt het belang-

rijkste concept op het gebied van gegevenstypen en heeft geleid tot een definitie waarmee de meesten het tegenwoordig eens zijn:

Gegevenstype. Een *gegevenstype* is een verzameling waarden en een aantal bewerkingen op die verzameling waarden.

Deze definitie is dezelfde als die welke wordt gebruikt voor de wiskundige structuur die *algebra* heet. Er zijn echter enkele belangrijke verschillen tussen gegevenstypen en algebra's; we zullen die in hoofdstuk 12 bespreken.

Er zijn nog andere belangrijke opvattingen over gegevenstypen ontwikkeld, bijvoorbeeld die van Smalltalk, Russell en SETL. En er is nog een andere belangrijke methode om typen te bekijken, namelijk die waarin wordt uitgegaan van een *universeel domein*. Vanuit deze optiek bestaat er slechts één fundamenteel domein van waarden. Typen geven alleen aan hoe die waarden moeten worden geïnterpreteerd. Deze aanpak sluit nauw aan bij de implementatie van gegevenstypen in programmeertalen. Het geheugen is een serie bits en de waarden van een gegevenstype worden voorgesteld als bepaalde bitpatronen. Het universele domein is de verzameling van alle bitpatronen; de typen geven aan hoe die bitpatronen moeten worden geïnterpreteerd. Zo kan het bitpatroon 01000001 worden geïnterpreteerd als het ASCII-teken 'A' of als het gehele getal 65. De interpretatie van het bitpatroon wordt bepaald door de operaties die op het type worden toegepast. In het algemeen geldt dat de interpretatie van een bitpatroon wordt bepaald door de operaties die erop worden toegepast. Door het aantal toegestane operaties in te perken, kan men ervoor zorgen dat de interpretatie van een bitpatroon overeenkomt met een bepaald 'type'. Een gegevenstype biedt dus een consistente manier om waarden uit het universele domein te interpreteren. Hoewel deze en andere zienswijzen in bepaalde situaties zinvol toe te passen zijn, gebruiken we in dit boek de algebraïsche zienswijze.

Discussies over gegevenstypen worden vaak vertroebeld door verwarrende terminologie – een situatie die in de wereld van de informatica en het programmeren vaak voorkomt. Wat voor de één een *object* is, is voor de ander een *waarde*. De *union* van de ene programmeertaal is het *record* van een andere programmeertaal. In de jaren zestig werd de nieuwe taal ALGOL 68 ontworpen en in de definitie kwamen veel nieuwe woorden voor die in de plaats van gebruikelijke, afgesleten woorden traden. Het was de bedoeling zo tot heel precieze betekenissen te komen in plaats van de gebruikelijke vage woorden te gebruiken met al hun ingebouwde connotaties. Die aanpak was ideaal voor formalisten die erop uit waren een precieze definitie te geven, maar voor de gemiddelde leek was

het een geweldig obstakel. Hoewel de literatuur over gegevenstypen nog niet consequent dezelfde terminologie hanteert, kunnen we wel de definities van een aantal termen voor dit boek vastleggen. De volgende definities verklaren de gebruikelijke woorden die informatici met betrekking tot gegevenstypen gebruiken.

Waarde. Een *waarde* is een wiskundige abstractie. Waarden zijn dus niet in ruimte en tijd gesitueerd. Ze kunnen niet in een computergeheugen worden bewaard of veranderd. Ze kunnen wel (via een codering) in het computergeheugen worden gerepresenteerd.

Operatie. Een *operatie* is een wiskundige functie op waarden,

Object. Een *object* is in ruimte en tijd gesitueerd en kan waarden hebben. Als een *object* een waarde heeft, wil dat zeggen dat een representatie van die waarde binnen dat object gecodeerd is. Gewoonlijk bestaat een object gedurende een bepaalde periode op een bepaalde plaats. De waarde van een object kan in de tijd veranderen.

Variabele. Een *variabele* is een object.

Symbool. In het kader van programmeertalen is een *symbool* een zelfstandig stukje tekst. Programma's zijn reeksen *symbolen* die algoritmen en computerinstructies weergeven.

Letterlijke constante (literal). Een *letterlijke constante* of *literal* is een symbool dat staat voor een waarde (een waarde denoteert). Er kan niet door herdefinitie een andere waarde aan worden toegekend. Letterlijke numerieke en stringconstanten worden het meest gebruikt.

Constante. De term *constante* wordt soms gebruikt in de betekenis van letterlijke constante, maar anders dan een letterlijke constante verwijst een constante meestal naar de waarde waar de letterlijke constante voor staat (de waarde die de letterlijke constante denoteert).

Identifier. Een *identifier* is een symbool dat wordt gebruikt om een waarde, een variabele, een operatie, een procedure of iets anders aan te duiden. Normaal is een identifier een serie alfanumerieke tekens, beginnend met een letter.

Operatorsymbool. Een *operatorsymbool* is een symbool dat wordt gebruikt om een operatie of een procedure aan te duiden. Operatorsymbolen hebben geen tevoren vaststaande betekenis. Niettemin gaan de meeste programmeertalen uit van de gebruikelijke betekenis van operatorsymbolen. Zo duidt '+' meestal de optellingsoperator aan.

Belangrijke kwesties met betrekking tot gegevenstypen

De verhandeling *Notes on Data Structuring* door C.A.R. Hoare (1972a) is een stimulerende en invloedrijke beschrijving van gegevenstypen, geschreven vóór de revolutie die door het abstracte gegevenstype is veroorzaakt. Hoare vat enkele belangrijke punten samen:

1. Een type bepaalt de klasse van waarden die een variabele of een expressie mag aannemen.

2. Elke waarde behoort tot één en slechts één type.

3. Het type van een waarde die wordt aangeduid door een constante, variabele of expressie kan worden afgeleid uit de vorm daarvan of uit de context, zonder enige kennis van de waarde zoals die bij uitvoering van het programma wordt berekend.

4. Elke operator verwacht operanden van een vast type en levert een resultaat van een vast type ... . Als hetzelfde operatorsymbool wordt toegepast op een aantal verschillende typen ..., kan dat symbool als dubbelzinnig worden beschouwd, omdat het in feite staat voor een aantal verschillende operatoren. Zo'n stelselmatige dubbelzinnigheid kan altijd al tijdens het compileren worden opgelost.

5. De eigenschappen van de waarden van een type en van de primitieve operatoren die erop zijn gedefinieerd, worden vastgelegd met behulp van een verzameling axioma's.

6. Informatie over typen wordt in hogere programmeertalen gebruikt om betekenisloze constructies in een programma te voorkomen en te ontdekken, en om de methode te bepalen volgens welke de gegevens in een computer gerepresenteerd en bewerkt moeten worden.

7. De typen waarin we geïnteresseerd zijn, zijn die waarmee wiskundigen al vertrouwd zijn, namelijk Cartesische produkten, discriminated unions, verzamelingen, functies, sequences en recursieve structuren.*

---

* Uit O.J. Dahl, C.A.R. Hoare, E.W. Dijkstra, *Structured Programming*, Academic Press, New York, 1972, pp. 92-93.

Deze beschrijving van gegevenstypen wekt misschien de indruk het laatste woord te zijn op dit gebied, maar moet eigenlijk meer worden beschouwd als een uitdaging. Voldoen gegevenstypen inderdaad aan deze beschrijving? Moeten gegevenstypen aan deze beschrijving voldoen? Deze vragen brengen ons tot enkele van de belangrijkste kwesties met betrekking tot gegevenstypen. Hoares eerste punt bijvoorbeeld is dan wel geen complete definitie, maar geeft de indruk dat een gegevenstype een verzameling waarden is. Zoals we al hebben besproken, vormt deze uitspraak een centraal deel van de definitie van een gegevenstype, maar de overige zes punten breiden de definitie en de rol van gegevenstypen nog uit. Het eerste punt herinnert ons er ook nadrukkelijk aan dat deze waarden en typen niet uit het niets verschijnen, maar worden afgeleid uit de variabelen en expressies van een echte programmeertaal.

Over het tweede punt op de lijst van Hoare valt te twisten. Mogen twee verschillende typen gelijke waarden bevatten? In de praktijk gebeurt dat bij de meeste gegevenstypen niet. Er bestaan echter vele belangrijke en interessante ontwikkelingen die uitgaan van een hiërarchie van typen, gebaseerd op subtypen en supertypen. Neem als eenvoudig voorbeeld de gehele getallen en de reële getallen. Als wiskundige abstractie is elk geheel getal een reëel getal. Precies zoals er vele manieren bestaan om getallen te noteren, zoals met Arabische of Romeinse cijfers of met mantisse en exponent, zo zijn er ook vele manieren om getallen in een computer te representeren. In PL/I kan het getal twee op vele verschillende manieren worden gecodeerd – bijvoorbeeld als floating, decimal of fixed binary. De programmeur heeft bij ieder numeriek type te doen met een verschillende representatie van het getal twee.

Punt 3 en 4 van de lijst van Hoare brengen het idee ter sprake om tijdens het compileren op typen te controleren. Het voor en tegen van typecontrole tijdens het compileren dan wel tijdens het uitvoeren van het programma vormt een belangrijk punt van discussie. De kern van de discussie is een verschil van mening over de vraag, van welk soort dingen nu precies het type moet worden gecontroleerd. Volgens één opvatting moet het type van *waarden* worden gecontroleerd, niet van *variabelen*. Dit soort typecontrole kan pas worden gedaan tijdens het uitvoeren van het programma, omdat de waarde van een variabele tijdens het compileren niet altijd bekend is. Verdergaande typecontrole tijdens het compileren kan worden bereikt als alle variabelen *getypeerd* zijn, dat wil zeggen van een type-aanduiding voorzien. De waarde van een getypeerde variabele moet altijd van één bepaald type zijn. Typecontrole wordt in de hoofdstukken 5 en 6 behandeld.

Het vijfde punt op de lijst van Hoare levert een middel om gegevenstypen te definiëren. Net als algebra's kunnen gegevenstypen met behulp van axioma's worden gedefinieerd. Een axioma is een uitspraak over de eigenschappen en kenmerken van waarden en operaties. Eén van de resultaten van de revolutie van de abstracte gegevenstypen is de aandacht die wordt geschonken aan het specificeren van typen en in het bijzonder aan het specificeren met behulp van axioma's. Een veel gebruikte vorm van deze methode wordt in hoofdstuk 12 en 13 beschreven.

Het zesde punt beschrijft een belangrijk doel van gegevenstypen. Het is interessant dat de kwestie van abstracte gegevenstypen in dit punt herkenbaar is. Het stelt vast dat de twee belangrijkste doelstellingen zijn: (1) typecontrole en (2) representatie en interpretatie.

Representatie. De *representatie* geeft aan hoe de waarden van een gegevenstype moeten worden gecodeerd.

Implementatie. Gegeven de representatie is de *implementatie* van een gegevenstype de verzameling algoritmen die de operaties op het gegevenstype implementeren.

Abstracte gegevenstypen. De belangrijkste gedachte achter *abstracte gegevenstypen* is de scheiding tussen het gebruik van een type en de representatie en implementatie ervan. Het gebruik van een type behoort alleen afhankelijk te zijn van de verzameling van waarden en operaties. Het behoort noch van de representatie, noch van de implementatie afhankelijk te zijn.

Abstracte gegevenstypen zijn waarschijnlijk de belangrijkste stap vooruit geweest die in de jaren zeventig op het gebied van programmeertalen is gezet. Abstracte gegevenstypen worden in hoofdstuk 8 en 9 behandeld.

Het laatste punt op Hoares lijst vormt een expliciete verwijzing naar de wiskundige structuren die als basistypen van de informatica worden gebruikt. Het overzicht van typen in hoofdstuk 2, 3 en 4 laat de historische ontwikkelingen zien die tot een dergelijke uitspraak hebben geleid.

In andere hoofdstukken van dit boek worden andere onderwerpen besproken, die niet expliciet door Hoare worden genoemd. Kwesties van opslag in het geheugen spelen een belangrijke rol bij gegevenstypen; die worden in hoofdstuk 7 behandeld. Hoofdstuk 7 beschrijft ook de relatie tussen waarden en variabelen

en tussen imperatief en applicatief programmeren. Ten slotte wordt in hoofdstuk 10 'polymorfisme' besproken.

1.4 Programmeertaalsyntaxis

Een groot probleem waarmee iedere auteur wordt geconfronteerd die problemen rond programmeertalen wil bespreken, is de keus van een programmeertaal om voorbeelden in te noteren. In dit boek is het belangrijk dat er voorbeelden worden besproken uit een grote verscheidenheid aan programmeertalen die van invloed zijn geweest op gegevenstypen. De feitelijke syntaxis van de taal is niet van belang, het gaat om de ideeën en concepten die door de taal zijn geïntroduceerd of uitgewerkt. Programmeertalen zijn berucht om hun uiteenlopende syntactische conventies. Het gebruik van vele programmeertalen zou in dit boek af en toe verwarrend kunnen worden. Om deze mogelijke verwarring kleiner te maken, zullen we een consistente programmeertaalsyntaxis aanhouden die op Ada* is gebaseerd. Waar dat nuttig is zullen we voorbeelden uit andere talen in deze standaardtaal vertalen. Soms maakt deze methode het nodig de semantiek van de gekozen taal te veranderen.

De keuze van een standaardprogrammeertaal is moeilijk geweest. Er bestaat een grote verscheidenheid aan conventies, elk met zijn eigen voor- en nadelen. Zo is er de ALGOL- en de Pascal-traditie voor declaraties:

`int X;`	ALGOL-traditie
`X: integer`	Pascal-traditie

De ALGOL-syntaxis heeft het voordeel van een eenvoudige en voor de hand liggende syntaxis voor initialisatie door middel van toevoeging van ":=3", terwijl de Pascal-syntaxis lijkt op de normale wiskundige notatie. Als het gaat om de typen record en union is de verscheidenheid soms zelfs verrassend voor wie aan een dergelijke verscheidenheid gewend is. Hier is bijvoorbeeld een lijstje van verschillende manieren om een record van twee gehele getallen te declareren.

`struct ( int x,y)`	ALGOL 68
`record x, y: integer end`	Pascal
`1 ITEM, 2 (X,Y) FIXED BINARY(15)`	PL/I

* Ada is een handelsmerk van het Department of Defense van de Verenigde Staten.

`DATA("ITEM(X,Y)")`	SNOBOL
`struct { int x,y; }`	C
`int#int`	Hope

Een taal kan nooit alle mechanismen en nuances bevatten van de gegevenstypen die in dit boek worden beschreven. Dus welke taal we ook kiezen voor de standaardsyntaxis, er zullen altijd tekortkomingen zijn, die we dan moeten opvangen door willekeurig te kiezen syntactische veranderingen of uitbreidingen. Zowel Ada als Pascal bieden een rijke syntaxis voor het noteren van gegevenstypen en abstracte gegevenstypen. Velen hopen dat Ada één van de belangrijke talen zal worden. Daarom is het redelijk een Pascal/Ada-achtige notatie te gebruiken voor het uitdrukken van ideeën: de meeste lezers hoeven dan geen nieuwe syntaxis te leren. Toch kunnen ook in Ada, evenmin als in enige andere programmeertaal, niet alle ideeën tot uitdrukking worden gebracht die we in dit boek willen behandelen.

Opgaven

1. Beschrijf een ander voorbeeld van een typensysteem dat in de maatschappij of in de wetenschap wordt gebruikt en geef voorbeelden van juist en onjuist gebruik ervan.

2. Onderzoek de structurele verschillen tussen classificatiesystemen voor bibliotheken.

3. Sommige typen worden gedefinieerd door eigenschappen. Andere typen worden expliciet opgebouwd zonder dat een verband met eigenschappen wordt gelegd. Bestudeer dit verschil aan de hand van de typensystemen die in dit hoofdstuk zijn genoemd.

4. Wat maakt het voor verschil of een gegevenstype wordt gedefinieerd als een verzameling waarden of als een verzameling waarden en operaties? Geef voorbeelden.

Literatuur

Het gebruik van gegevenstypen in de informatica vindt zijn oorsprong in de wiskunde. Bertrand Russell heeft in 1908 in zijn artikel ‘Mathematical logic

based on the theory of types' een streng typensysteem voor wiskundige systemen geschapen om paradoxen zoals de paradox van Russell te omzeilen. Het werken aan typen is daarna verder gegaan, parallel aan de ontwikkelingen in de formalisering van de wiskunde en de logica. De door Alonzo Church in de jaren dertig ontwikkelde lambda-calculus is gebruikt als eenvoudige, maar niet-triviale taal voor het ontwikkelen van typensystemen en -theorieën, waarvan de meeste in onmiddellijk verband staan met programmeertalen. Zowel Hoare (1972) als deel IV uit Gries (1976) zijn bronnen van inspiratie geweest voor de stof in paragraaf 1.2. Scott (1976) geeft een elegante wiskundige formulering van het gezichtspunt van het universele domein. De programmeertaal Russell is een belichaming van datzelfde gezichtspunt; zie Demers et al. (1978, 1980).

Deel I

Een overzicht van gegevenstypen

2

Basistypen

Gegevenstypen in programmeertalen zijn soms gemakkelijk, soms moeilijk te begrijpen. Soms denken we een programmeertaal goed te begrijpen en komen dan ineens iets onverwachts tegen. Zelfs eenvoudige typen, zoals getallen, kunnen ingewikkeld zijn. In verschillende talen zijn typen, zelfs als ze dezelfde naam hebben, vaak op kleine, maar belangrijke punten verschillend. De complexiteit van veel typen en veel talen vormt een uitdaging voor wie systeem wil brengen in gegevenstypen. Er is zelfs geen algemeen aanvaarde definitie van het begrip gegevenstype.

Voordat we ons bezig houden met algemene eigenschappen van gegevenstypen, is het nuttig eerst de grote verscheidenheid aan gegevenstypen te bestuderen die we in programmeertalen tegenkomen. Daarom geven de nu volgende drie hoofdstukken een overzicht van de gegevenstypen die zijn ontstaan sinds de tijd voor FORTRAN tot de tegenwoordige tijd van Ada. We beginnen in dit hoofdstuk met de eenvoudigste gegevenstypen, de basistypen. De basistypen zijn de typen die niet zijn opgebouwd uit andere typen; voorbeelden zijn Booleaanse, teken-, opgesomde en getalstypen. De overige typen, soms *constructortypen* genoemd, worden opgebouwd met behulp van andere typen; ze worden behandeld in de hoofdstukken 3 en 4.

2.1 Booleaanse typen

Het eenvoudigste gegevenstype is het Booleaanse type. De naam is afgeleid van die van de Engelse wiskundige George Boole, die als eerste algebra's van

twee elementen heeft onderzocht. Het Booleaanse gegevenstype heeft twee waarden, van oudsher waar en onwaar genoemd. Booleaanse waarden leveren de meest gebruikte manier om met behulp van conditionele en herhalingsopdrachten de voortgang van een programma te besturen. Booleaanse typen zijn in besturingsopdrachten niet per se nodig, omdat de syntaxis van besturingsopdrachten ook expressies kan toestaan van de vorm:

expressie   vergelijkingsoperator   expressie

Vele talen gebruiken het woord Boolean als naam voor het type, en de identifiers *true* en *false* als constanten. Sommige talen korten Boolean af tot Bool, maar andere hebben volkomen verschillende namen; zo heeft FORTRAN het type LOGICAL en PL/I het type BIT. In deze talen worden de constanten true en false anders aangegeven. In FORTRAN wordt true weergegeven als .TRUE. en in PL/I als '1'B. Sommige talen hebben niet eens een apart Booleaans type, maar maken gebruik van een ander type. In LISP bijvoorbeeld staat de waarde NULL voor false en staan alle andere waarden voor true. De taal C gebruikt nul voor false en alle andere waarden voor true. Het is onder programmeurs in die talen de gewoonte om constanten of macro's true en false te definiëren om de programma's leesbaarder te maken.

Er zijn precies vier Booleaanse functies met één argument en 16 met twee argumenten. De vier met één argument zijn de twee constante functies, de identieke functie en de ontkenningsfunctie. De ontkenningsfunctie levert de andere waarde als resultaat. De meeste talen beschikken over deze functie, maar iedere taal lijkt die met een verschillend symbool aan te duiden (bijvoorbeeld not, .NOT., !, -, ~, ¬ en /). Van de 16 functies met twee argumenten zijn gewoonlijk de functies and en or beschikbaar, ook weer aangeduid met uiteenlopende symbolen. Met behulp van deze drie functies *not*, *and* en *or* kunnen alle Booleaanse functies worden gedefinieerd. Om de uitwerking van expressies in de hand te houden zijn de Booleaanse conditionele operatoren *andif* en *orif* nuttig. Ada gebruikt de term `and then` voor *andif* en `or else` voor *orif*. Deze operatoren leveren hetzelfde resultaat als de operatoren *and* en *or*, maar het tweede operand wordt alleen uitgewerkt als dat werkelijk nodig is voor het bepalen van de uitkomst. Als we uitgaan van de axioma's

true or X = true
false and X = false

zien we dat het tweede operand in bepaalde situaties niet uitgewerkt hoeft te worden. Het gebruik van zulke conditionele operatoren heeft twee voordelen.

Ten eerste kunnen expressies zo sneller worden uitgewerkt. Ten tweede wordt het mogelijk bepaalde condities in één expressie uit te drukken, zoals in:

```
if J>ONDERGRENS and J<BOVENGRENS then
        if DATAVECTOR(J)≠0 then
                UITKOMST := 1/DATAVECTOR(J);
        else
                MELD_FOUT();
        end if;

else
        MELD_FOUT();
end if;
```

In de bovenstaande situatie kan niet met één conditionele expressie met and worden volstaan, omdat de index van de array dan altijd zou worden uitgewerkt, waarbij de index buiten de arraygrenzen kan vallen. Met behulp van conditionele operatoren kan de expressie vereenvoudigd worden tot

```
if J>ONDERGRENS and J<BOVENGRENS
                and then DATAVECTOR(J)≠0 then
        UITKOMST := 1/DATAVECTOR(J);
else
        MELD_FOUT();
end if;
```

2.2 Tekentypen

De typen teken en string worden vaak verward omdat er in veel talen geen onderscheid tussen wordt gemaakt. Een teken is een ondeelbaar symbool. Een string is een reeks van nul of meer tekens. Soms is het nuttig op dit verschil tussen tekens en strings te letten, omdat het in zekere zin even belangrijk is als het verschil tussen integers en arrays van integers. Een teken is niet gewoon maar een string van lengte 1, net als een integer niet gewoon een array van lengte 1 is. Anders dan met integers is er met afzonderlijke tekens niet veel te doen, terwijl er met strings juist veel kan worden gedaan. Daarom kennen sommige talen alleen strings.

Net als getallen is het type teken vaak machine-afhankelijk, dat wil zeggen iedere machine heeft zijn eigen verzameling tekens en een code voor elk teken. Bijna alle IBM-machines gebruiken Extended Binary Coded Decimal Interchange Code (EBCDIC, uitgebreide binair gecodeerde decimale uitwisselingscode) en bijna alle niet-IBM-machines gebruiken American Standard Code for

Information Interchange (ASCII, Amerikaanse standaard-code voor informatie-uitwisseling). Het verschil wordt meteen duidelijk als u een lijst getallen en namen sorteert. Sorteren berust op de vergelijkingsoperatie. Ieder teken heeft een unieke tekencode. Het vergelijken van tekens is gebaseerd op die tekencodes. De volgorde van de letters is in ASCII en EBCDIC gelijk (namelijk A<B<C...), maar voor de andere tekens is de volgorde verschillend. Het teken 0 is bijvoorbeeld op EBCDIC-machines groter dan A, terwijl op ASCII-machines A groter is dan 0. Sorteeralgoritmen die gebaseerd zijn op tekencodes geven op machines met verschillende tekencodes een verschillende resultaat. Voorbeelden van andere machine-afhankelijke operaties zijn de functies die tekens naar integers converteren en andersom. Deze conversiefuncties converteren tekens eenvoudig naar tekencodes en andersom. Een gebruikelijke methode om op ASCII-machines de opvolger van een alfabetische teken te krijgen is de expressie

```
INT_TO_CHAR( 1 + CHAR_TO_INT(X) )
```

waarin INT_TO_CHAR en CHAR_TO_INT de conversiefuncties zijn. Dit werkt niet op EBCDIC-machines, omdat de EBCDIC-codes voor het alfabet niet aaneensluitend zijn. In de tabel hieronder is te zien dat de tekens '0' en 'A' niet in dezelfde volgorde voorkomen en dat de codes voor de letters 'I' en 'J' niet op elkaar volgen.

Enkele tekencodes:

Teken	*ASCII*	*EBCDIC*
0	48	240
A	65	193
I	74	201
J	75	209

2.3 Opgesomde typen

Een elegante en eenvoudige generalisatie van de tot nu toe genoemde typen is het *opgesomde type*. Een opgesomd type is een eindige verzameling waarden, gewoonlijk gespecificeerd door een lijst waarden of met behulp van een onder- en een bovengrens binnen een eerder gedefinieerd opgesomd type. De tweede methode van specificeren levert een *deelbereik* of *subrange*. Opgesomde typen

en deelbereiken zijn voor het eerst in Pascal gebruikt. De typen teken en Boolean en de numerieke typen kunnen alle worden beschouwd als opgesomde typen.

Programmeurs kunnen opgesomde typen introduceren voor waarden van andere typen dan getallen, tekens of Booleaanse waarden. Zulke opgesomde typen kunnen worden gebruikt voor vele verschillende soorten gegevens. De volgende twee typen zijn daar voorbeelden van:

```
type DEUR is (OPEN, OP_EEN_KIER, DICHT, AFGESLOTEN);
type KLEUR is (ROOD, ORANJE, GEEL, GROEN, BLAUW, PAARS);
```

De symbolen OPEN tot en met AFGESLOTEN zijn letterlijke constanten van het type DEUR. Opgesomde typen lijken misschien ongewoon, maar ze zouden waarschijnlijk tot de meest gebruikte typen behoren als ze maar beschikbaar waren en steeds op de juiste momenten toegepast werden. Vaak zal een programmeur met vastgeroeste gewoonten een representatie met behulp van integers kiezen in situaties waar een opgesomd type duidelijk beter is uit het oogpunt van documentatie en voorkoming van fouten.

Deelbereiken of subranges zijn ook opgesomde typen; ze worden gedefinieerd door het opgeven van onder- en bovengrens in een eerder gedefinieerd gebied. In Pascal-achtige talen worden getalstypen gewoonlijk gespecificeerd als een deelgebied van een vast numeriek type. De grenzen van de subrange worden meestal gescheiden door het symbool '..', zoals in de volgende twee voorbeelden:

```
type JAAR is 1900..1999;
type ONAFGESLOTEN_DEUR is OPEN..DICHT;
```

Ada heeft onder andere de volgende operatoren op opgesomde waarden:

FIRST levert het eerste element van de opsomming
LAST levert het laatste element van de opsomming
SUCC(X) levert het element dat op X volgt (successor)
PRED(X) levert de voorganger van X (predecessor)

Opsommingen leveren ook een natuurlijke volgorde die kan worden gebruikt voor het definiëren van vergelijkingsoperatoren.

2.4 Getallen

Getallen zijn tegelijk de eenvoudigste en de moeilijkste van alle gegevenstypen. Ze zijn het eenvoudigst omdat iedereen vertrouwd is met getallen. Maar ze zijn het moeilijkst omdat computers getallen op veel verschillende manieren representeren; vaak is het nodig de details daarvan te kennen om zelfs maar het eenvoudigste programma te kunnen schrijven. Het hele idee van gegevenstypen is ontstaan als gevolg van verschillende representaties voor getallen. Op de vroegste computers, vóór de opkomst van hogere programmeertalen en zelfs van assembleertalen, werden getallen op allerlei manieren voorgesteld. Binaire en decimale machines representeren getallen verschillend. Er zijn ook machines die beide representaties mengen door binair gecodeerde decimalen te gebruiken. Er is geen eindige representatie die gebruikt kan worden om alle integers of alle reële getallen voor te stellen. Het is alleen mogelijk een deelverzameling te representeren. De omvang van die deelverzameling hangt af van het aantal bits dat voor de representatie wordt gebruikt. Die omvang wordt gewoonlijk uitgedrukt in het aantal cijfers precisie. Op sommige machines kan gerekend worden met verschillende representaties, die dan alleen verschillen in het aantal cijfers precisie. De manier waarop de decimale punt wordt behandeld kan ook van representatie tot representatie verschillen.

De twee methoden die we in meer detail zullen bekijken zijn de representaties met *fixed-point* (vaste komma) en met *floating-point* (drijvende komma). Deze verschillende methoden vormden al ver vóór de opkomst van de programmeertalen een krachtig en ingewikkeld typensysteem. Ze geven ons ook inzicht in de verschillen tussen typecontrole tijdens het compileren en controle tijdens het uitvoeren van een programma. Het is tegenwoordig een algemeen aanvaarde opvatting dat typecontrole tijdens compilatie beter is dan tijdens de uitvoering, maar getallen in floating-point-representatie leveren een aardig tegenvoorbeeld.

De exponent van een getal kan beschouwd worden als het type van het getal. Bij de fixed-point-notatie worden de exponenten berekend voordat het programma wordt uitgevoerd; bij het werken met floating-point worden de exponenten tijdens het uitvoeren van het programma berekend. Om de uitwerking te zien van deze voorloper van de gegevenstypen uit programmeertalen is het leerzaam eens te kijken naar de werkzaamheden van een programmeur in de jaren vijftig.

Laten we uitgaan van een machine die getallen representeert als acht decimale cijfers en een tekenbit. Er is geen expliciete decimale punt; die wordt veronder-

steld rechts van het meest rechtse cijfer te staan. Zo stelt '+00001984' het getal 1984 voor. De rekenkundige bewerkingen optellen, aftrekken, vermenigvuldigen en delen gaan uit van integers. Het produkt van twee getallen wordt voorgesteld als een tweetal getallen, waarvan het eerste de overflow (of overloop) bevat, bijvoorbeeld:

+ 12345678	eerste operand maal
+ 00000100	tweede operand
+ 00000012	eerste getal van uitkomst
+ 34567800	tweede getal van uitkomst

De deling levert een quotiënt en een rest, bijvoorbeeld:

+ 00000500	deeltal
+ 00000003	deler
+ 00000166	quotiënt
+ 00000002	rest

De machine lijkt erg beperkt te zijn door deze getalsrepresentatie met deze verzameling van operaties, maar in de jaren vijftig was dit heel normaal. We kunnen de kracht en het nut van de representatie beduidend opvoeren door ons de decimale punt gewoon ergens anders voor te stellen dan geheel rechts van het getal. Het is dan wel nodig goed op te passen en geen fouten te maken. Precies zoals de gebruiker van een rekenlineaal zelf de plaats van de decimale punt moest bijhouden, zo moest ook de programmeur van deze machine zelf de plaats van de punt bijhouden. Er bestonden verscheidene methoden die de programmeur daarbij konden helpen.

Een heel simpele methode is de decimale punt een vaste plaats te geven en alle getallen te representeren met de fictieve decimale punt op die plaats. Men kan bijvoorbeeld het midden van het getal van acht cijfers als plaats voor de decimale punt kiezen. Dan stelt '+00025000' het getal twee en een half voor. De operaties optellen en aftrekken blijven bij deze representatie correct werken. Maar in de bewerkingen vermenigvuldigen en delen zijn nu veranderingen nodig. Het resultaat van een vermenigvuldiging moet over vier decimale plaatsen worden verschoven. Als twee en een half wordt vermenigvuldigd met drie (voorgesteld als '+00030000'), dan bestaat het resultaat uit het tweetal '+00000 007' en '+50000000'. Pas nadat we dit tweetal over vier posities hebben verschoven, krijgen we het gewenste resultaat '+00075000'. Voor de deling is een soortgelijke regeling nodig.

Door nu deze representatie in het gehele programma te gebruiken, en na elke vermenigvuldiging en elke deling te schuiven, kunnen we deze machine, die eigenlijk alleen met integers van acht cijfers werkt, gebruiken als een machine die rekent met acht cijfers waarvan vier rechts van de decimale punt. De programmeur zal de gewone integers nog nodig hebben voor het adresseren en voor invoer, uitvoer en indexering. Daardoor zullen in de meeste programma's zowel de fictieve getallen als de standaardgetallen voorkomen. De machine weet het verschil niet. Om correcte programma's te ontwikkelen moet de programmeur zelf nauwkeurig bijhouden welke getallen 'normaal' zijn en welke volgens de nieuwe representatie moeten worden geïnterpreteerd. Dat is een vorm van typecontrole die met de hand wordt uitgevoerd. Af en toe is het nodig een getal van de ene representatie in de andere te converteren. Die conversie is dan gewoon een verschuiving naar links of naar rechts over vier decimale plaatsen. Bij die conversie zullen zich af en toe fouten voordoen, omdat in geen van beide representaties alle getallen uit de andere representatie kunnen worden voorgesteld.

De boven beschreven methode verschaft ons geen erg groot bereik van getallen. Om meer flexibiliteit te krijgen kunnen we een *schaalfactor* gebruiken voor het bijhouden van de plaats van de decimale punt. Bij die methode wordt elk getal voorzien van een schaalfactor die de plaats van de decimale punt aangeeft. De programmeur bepaalt voordat hij gaat programmeren de schaalfactor voor elk getal dat voorkomt in de invoer, in de uitvoer of in de bewerkingen daartussen. De schaalfactor wordt gedefinieerd als de macht van tien waarmee we het getal moeten vermenigvuldigen om het in feite gerepresenteerde getal te krijgen. Als de schaalfactor 0 is, betekent dat, dat de decimale punt geheel rechts staat. Een schaalfactor – 4 betekent dat de decimale punt in het midden van het getal staat. Voor het getal twee en een half zijn er nu zeven verschillende representaties:

Representatie	*Schaalfactor*	*Waarde*
+ 00000025	– 1	2.5
+ 00000250	– 2	2.5
+ 00002500	– 3	2.5
+ 00025000	– 4	2.5
+ 00250000	– 5	2.5
+ 02500000	– 6	2.5
+ 25000000	– 7	2.5

We kunnen een getal van de ene representatie in de andere overvoeren door het te schuiven. Alleen getallen met dezelfde schaalfactor kunnen worden opgeteld

of afgetrokken. Het is niet mogelijk het getal '00000025' met schaalfactor 1 (waarde 250) op te tellen bij het getal '00000005' met schaalfactor 2 (waarde 500); daarvoor is het nodig eerst een van de operanden anders te schalen.

Representatie	*Schaalfactor*	*Waarde*	
+ 00000025	1	250	
+ 00000005	2	500	
+ 00000030	?	?	Som zonder herschaling
+ 00000007	2	700	Som na herschaling van eerste operand
+ 00000075	1	750	Som na herschaling van tweede operand

Zoals uit het voorbeeld blijkt, is het, om verlies van precisie te voorkomen, belangrijk het juiste operand te herschalen. Maar omdat er ook overflow kan optreden, kan het nodig zijn toch in de andere richting te herschalen. Bij vermenigvuldigen is herschaling van de operanden niet nodig; de schaalfactor van het resultaat is de som van de schaalfactoren van de operanden.

Toepassing van de schaalfactorenmethode omvat de volgende stappen:

1. Bepaal uit gegevens over de fysieke grootheden de maximale grootte van alle getallen ... Bepaal in geval van deling ook de minimale waarde van de delers of de maximale waarde van de quotiënten. Hiervoor zal gewoonlijk kennis nodig zijn van invoer, tussenresultaten en uitvoer; op zijn minst moet de informatie over de invoer beschikbaar zijn.

2. Leg de relatie tussen de echte getallen en de geschaalde versies vast door de noodzakelijke schaalfactoren op te schrijven. Die zullen gewoonlijk gelijk zijn aan de macht van tien die net groter is dan de maximale waarde van de betreffende grootheid. Dus als x in een bepaald probleem nooit groter kan worden dan 100, dan is de schaalfactor 2 ...

3. Substitueer de geschaalde grootheden in de vergelijkingen van het probleem. Verreken exponenten met elkaar en streep ze waar mogelijk tegen elkaar weg.

4. Grootheden die moeten worden opgeteld of afgetrokken, moeten gelijke schaalfactoren hebben. Als die voorwaarde voor een bepaalde vergelijking nog niet geldt, moeten, voordat de optelling of aftrekking plaatsvindt, sommige schaalfactoren door schuiven in de betreffende getallen worden veranderd. Het aantal posities

waarover moet worden geschoven is gelijk aan het verschil van de beide betrokken schaalfactoren.

5. Als bij een deling de deler na verrekening van de schaalfactoren nog een schaalfactor ongelijk 0 heeft, dan moet het resultaat naar rechts worden geschoven. Bij een vermenigvuldiging houden overgebleven schaalfactoren in, dat het produkt naar links moet worden geschoven, waardoor aan de linkerkant geen verlies van significante cijfers wordt veroorzaakt ...*

Deze regels voor typecontrole kunnen kort en formeel worden uitgedrukt. Als *x[n]* staat voor het getal *x*, gerepresenteerd met schaalfactor *n*, dan geven de volgende regels een overzicht van de typecontrole (<< en >> staan voor schuiven naar links respectievelijk rechts).

$$x[n] + y[n] = (x + y)[n]$$
$$x[n] - y[n] = (x - y)[n]$$
$$x[n] * y[m] = (x * y)[n + m]$$
$$\mathrm{x}[n] \,/\, y[m] = (x \,/\, y)[n - m]$$
$$\mathrm{x}[n] \ll m = \mathrm{x}[n - m]$$
$$\mathrm{x}[n] \gg m = \mathrm{x}[n + m]$$

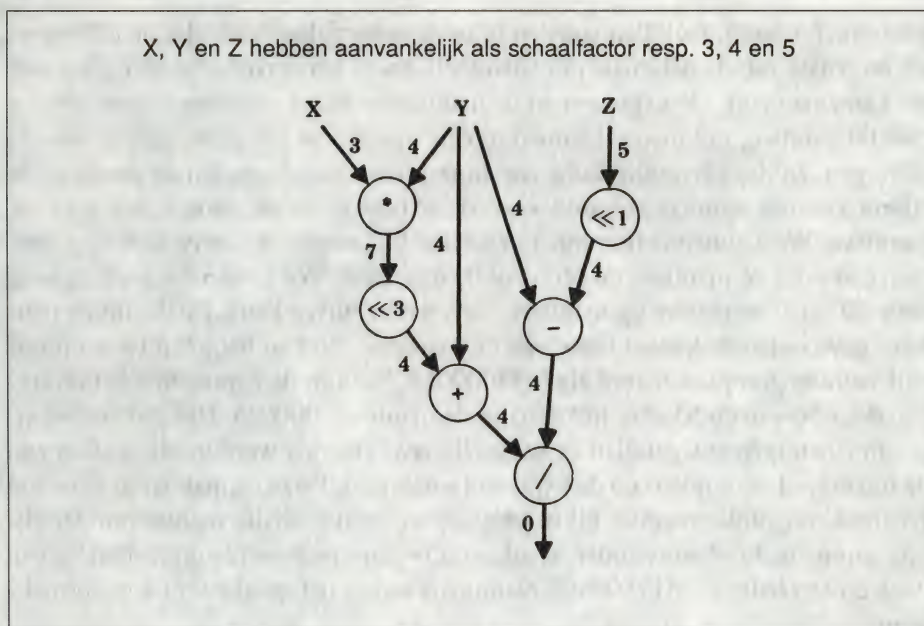
Figuur 2-1 laat een voorbeeld zien met drie grootheden als invoer en één als uitvoer. De getallen geven de schaalfactoren van de invoer- en uitvoergrootheden aan, alsmede van de tussenresultaten.

De getallen met schaalfactoren vormen een systeem van gegevenstypen. Elke schaalfactor *n* genereert een gegevenstype met zijn eigen verzameling waarden en operaties.

De boven beschreven stappen geven aan hoe de typen (dat wil zeggen de schaalfactoren) moeten worden bepaald, hoe typecontrole bij optellen en aftrekken moet worden uitgevoerd en hoe conversie van het ene type naar het andere kan gebeuren. Al deze stappen moeten met de hand uitgevoerd en gedocumenteerd worden. Daar kunnen op verschillende manieren fouten bij gemaakt worden. Schrijffouten bij het controleren van de typen (zoals fouten bij het verrekenen van twee exponenten) kunnen later ongeldige optellingen of verkeerd geschaalde resultaten tot gevolg hebben.

---

* Uit D.D. McCracken, *Digital Computer Programming*, John Wiley and Sons, Inc., New York, 1957.



*Figuur 2-1 Stroomdiagram van "(X*Y + Y) / (Y – Z)" met schaalfactoren.*

Een verkeerde schatting van de minimale of maximale grootte van invoergrootheden kan overflow of underflow veroorzaken. Nu kunnen schrijffouten niet automatisch worden gedetecteerd, maar voor het tweede geval kan bij elke vermenigvuldiging, deling en conversie gecontroleerd worden op overflow of underflow, zodat het systeem een fout kan ontdekken tijdens de uitvoering van het programma. We hebben een derde bron van fouten doordat afkapping leidt tot resultaten die slechts bij benadering juist zijn. Numerieke wiskunde is een gebied van de wiskunde dat kan worden gebruikt om de foutenmarges te bepalen die bij fixed-point- en floating-point-aritmetiek voorkomen.

In veel toepassingen is het moeilijk te schatten hoe groot of klein de numerieke in- of uitvoer zal zijn. Vaak is het gebied te groot om precieze resultaten voor het gehele gebied mogelijk te maken. In zulke gevallen nemen programmeurs vaak hun toevlucht tot rekenen met een *floating-point*. Bij de methode met schaalfactoren worden alle schaalfactoren vóór het coderen van het programma bepaald. Zulke methoden worden vaak *fixed-point*-methoden genoemd, omdat de plaats van de decimale punt onveranderd blijft. Bij de floating-point-methode worden de schaalfactoren tijdens de berekeningen bepaald. Deze aanpak maakt het nodig dat ieder getal tijdens de uitvoering van het programma een

schaalfactor heeft. Getallen worden in twee onderdelen verdeeld, de *exponent*, die de plaats van de decimale punt aangeeft, en de *mantisse**, die de cijfers van het getal aangeeft. De exponent en de mantisse worden gerepresenteerd als een tweetal getallen, dat meestal samen in één woord van het geheugen wordt opgeborgen. In deze hypothetische machine bijvoorbeeld zouden de eerste twee cijfers kunnen worden gebruikt voor de exponent en de andere zes voor de mantisse. We hebben echter maar één tekenbit, terwijl we er twee nodig hebben: één voor de mantisse en één voor de exponent. We lossen dat probleem op door 50 bij de exponent op te tellen – een willekeurige keus. De decimale punt staat gewoonlijk helemaal links van de mantisse. Zo kan het getal twee en een half worden gerepresenteerd als '+55000025', waarin de exponent 5 is (verkregen door 50 van de 55 af te trekken) en de mantisse .000025. Het is de gewoonte om floating-point-getallen te *normaliseren*. Daarbij worden alle nullen aan de linkerkant verwijderd en de exponent aangepast. Deze aanpak zorgt voor een zo groot mogelijke precisie bij berekeningen, omdat op die manier minder cijfers aan de rechterkant worden afgekapt. De genormaliseerde representatie van twee en een half is '+51250000'. Natuurlijk wordt het getal nul niet genormaliseerd.

De operaties op floating-point-getallen zijn ingewikkelder dan die op fixed-point getallen. Zowel de mantisse als de exponent moeten worden berekend. Voor optellen en aftrekken moet het systeem één van beide operanden herschalen. Al deze complicaties blijven verborgen binnen de operaties, zodat de programmeur zich niet met de details van de schaalfactoren hoeft bezig te houden. Het computersysteem levert subroutines of speciale hardware. Het is zinloos deze routines te gebruiken voor andere getalsrepresentaties. De programmeur moet dus nog steeds zelf het type van een grootheid bepalen (floating-point of fixed-point, en in het laatste geval de schaalfactor) en hij moet zorgen voor het controleren en converteren van de typen.

Er zijn nog andere getalsrepresentaties dan fixed-point en floating-point. Sommige representaties zijn alleen geschikt voor positieve getallen. Ook als het teken van het getal wel wordt opgenomen, zijn er vele verschillende representaties mogelijk, onder andere *teken en grootte*, *tweecomplement* en *eencomplement*. Verder is er steeds sprake van een compromis tussen nauwkeurigheid en kosten. Hoe groot moet de precisie zijn bij het rekenen? Het is meestal beter enkele extra cijfers te hebben, maar dit brengt meer kosten met zich mee. Voor

---

* De term *mantisse* is ontleend aan de rekenkunde en betekent het decimale deel van een gewone logaritme.

het opslaan van de cijfers is meer geheugen nodig en voor het uitvoeren van de operaties is meer tijd nodig. Naast decimale machines zijn er ook binaire machines. Sommige machines beschikken, om alle klanten tevreden te stellen, over vele representaties.

Getallen in FORTRAN

In het midden van de jaren vijftig werd bij IBM door een team onder leiding van John Backus de programmeertaal FORTRAN ontwikkeld en geïmplementeerd. Het doel van het FORTRAN-project was het automatiseren van het werk van de programmeur. Het was de bedoeling ingenieurs en wetenschappers in staat te stellen hun programma's in de taal FORTRAN te specificeren, waarna de FORTRAN-compiler het gehele programmeerwerk zou doen. Een compiler is een programma dat een programma geschreven in een hogere programmeertaal als FORTRAN, vertaalt in assembleertaal of machinetaal. Een deel van het werk van de FORTRAN-compiler is het automatisch controleren van de getalstypen, net als een menselijke programmeur dat zou doen. Sommigen dachten dat talen zoals FORTRAN spoedig de meeste programmeursbanen zouden doen verdwijnen. In plaats daarvan werd het beroep van de programmeur verbreed. Men bedacht en implementeerde grotere en complexere programma's en hield zich bezig met belangrijker kanten van het ontwerpen van programma's, in plaats van alleen met het coderen. Talen als FORTRAN verhoogden de produktiviteit. Maar de verhoogde vraag naar software overtrof de verhoogde produktiviteit, waardoor het aantal programmeursbanen gestaag is gegroeid in plaats van afgenomen.

De taal FORTRAN is als eerste programmeertaal belangrijk geweest in de geschiedenis van programmeertalen. De numerieke typen van FORTRAN zijn een afspiegeling van de van oudsher gebruikte typen, op de fixedpoint typen na. In tegenstelling tot FORTRAN voeren machine- en assembleertalen geen typecontrole uit en daarom kan een getal, in welke representatie dan ook, gewoonlijk in elk register en in elke geheugenplaats worden opgeslagen. In FORTRAN is het begrip geheugenplaats geabstraheerd tot een *variabele*. Variabelen worden in FORTRAN met identifiers aangeduid, die tot zes tekens lang kunnen zijn. Elke variabele staat voor een geheugenplaats en bij elke variabele hoort een type dat aangeeft welk soort getallen er in die geheugenplaats kunnen worden bewaard. Andere soorten getallen kunnen in die geheugenplaats niet worden opgeslagen. Die beperking wordt door de taal afgedwongen. De programmeur kan het type van elke variabele aangeven. Dat kan expliciet gebeuren in

een declaratie. Als van een variabele geen expliciete declaratie wordt gegeven, bepaalt de eerste letter van de identifier het type. Dan geeft een identifier die met een letter tussen de I en de N begint, een integer variabele aan en alle andere identifiers floating-point-variabelen. De programmeur kan die regel door middel van declaraties omzeilen.

De informatie over de typen dient verschillende doeleinden.

1. De compiler gebruikt de type-informatie om te bepalen welke machine-instructie er moet worden gebruikt. Dat maakt het programmeren veel gemakkelijker, omdat een programmeur in FORTRAN het symbool '+' kan gebruiken voor het optellen van elk soort getallen. Het is één van de taken van de FORTRAN-compiler om het plus-teken te *vertalen* in de geëigende optelopdracht van de computer. Deze vertaling wordt *operatoridentificatie* genoemd en kan alleen gedaan worden op basis van de typen van de operanden.
2. De compiler gebruikt de type-informatie ook om vast te stellen hoeveel geheugen er voor elke variabele moet worden ingeruimd. Verschillende representaties voor getallen kunnen verschillende hoeveelheden geheugen eisen.
3. Type-informatie helpt de compiler ook bij het ontdekken van fouten. Een *typefout* is het onjuiste gebruik van een waarde van een bepaald gegevenstype. (Met een typefout wordt in dit boek dus geen tikfout bedoeld.) Het controleren of alle waarden gebruikt worden op een manier die in overeenstemming is met het type, wordt *typecontrole* genoemd. Als de programmeur probeert een integer bij een real op te tellen, dan ziet de compiler daar een fout in, omdat computers in het algemeen geen optelopdracht voor gemengde typen hebben.

Sommigen vonden deze foutencontrole in bepaalde gevallen hinderlijk. Het is soms noodzakelijk om integers en reals door elkaar te gebruiken. In machine- of assembleertaal wordt aan dat verlangen tegemoet gekomen door middel van een aanroep van een conversieroutine, die een getal dat als integer is gerepresenteerd, overvoert in een getal dat als real is gerepresenteerd. In FORTRAN zijn FLOAT en IFIX de ingebouwde conversiefuncties. In de oorspronkelijke versie van FORTRAN waren gemengde expressies verboden, omdat men "meende dat, als er code voor typeconversie moest worden gegenereerd, de gebruiker zich daarvan bewust moest zijn, en dat de beste manier om hem ervan bewust te laten zijn, erin bestond te eisen het zelf aan te geven"*. Aan sommige

---

* John Backus (1978a)

versies van FORTRAN zijn automatische conversies toegevoegd die integers naar reals converteren, waardoor expressies met gemengde typen legaal werden.

FORTRAN IV heeft vier numerieke typen: INTEGER, REAL, DOUBLE PRECISION en COMPLEX. DOUBLE PRECISION wordt gebruikt als er voor reals meer cijfers precisie nodig zijn. In sommige versies van FORTRAN geeft '*n' na INTEGER, REAL of COMPLEX aan hoeveel bytes geheugen voor een getal gebruikt moeten worden (*n* is het aantal bytes).

Het *domein* (range) van een numeriek type is de verzameling getallen die erdoor kunnen worden gerepresenteerd. FORTRAN en de meeste andere programmeertalen leggen het domein van numerieke typen niet vast. De implementaties van de taal bepalen het domein van elk numeriek type. Om op verschillende machines efficiënt te zijn kan elke implementatie verschillende domeinen hebben. Domeinen zijn *machine-afhankelijk*, hetgeen betekent dat programma's die voor de ene machine zijn geschreven op de andere machine een verschillend resultaat kunnen hebben. Als de ene machine bijvoorbeeld grotere woorden heeft dan een andere, dan is het minder waarschijnlijk dat daarop overflow optreedt. De mogelijkheid hetzelfde programma op verschillende machines uit te voeren met hetzelfde resultaat, is een hoog gewaardeerde en begerenswaardige eigenschap en wordt *portabiliteit* of *overdraagbaarheid* genoemd, omdat het programma dan naar andere machines overgedragen kan worden. Het feit dat het zo moeilijk is, en daardoor inefficiënt, om te proberen getalsrepresentaties op alle machines gelijk te maken, heeft tot gevolg dat het voorlopig niet waarschijnlijk is dat numerieke gegevenstypen machineonafhankelijk worden, maar het werk aan standaards, zoals de standaard van IEEE voor floating-point typen, kan op de langere termijn van invloed zijn.

Getallen in PL/I

Een van de ingewikkeldste numerieke typensystemen is te vinden in PL/I. Ook PL/I is ontworpen bij IBM. De taal was bedoeld om voor alle toepassingen geschikt te zijn. Om deze algemene toepasbaarheid te bereiken wilden de ontwerpers de gebruiker een zo groot mogelijke flexibiliteit toestaan bij het kiezen van getalsrepresentaties. In PL/I worden gegevenstypen gespecificeerd met behulp van één of meer *attributen*. De attributen voor de numerieke typen van PL/I zijn:

1. FIXED, FLOAT of COMPLEX.
2. BINARY of DECIMAL.
3. SIZE (bepaalt het aantal cijfers precisie en de plaats van de decimale punt).

De programmeur kan deze attributen op vele manieren combineren. In tegenstelling tot FORTRAN beschikt PL/I ook over fixed-point typen (natuurlijk automatisch, op het bepalen van de domeinen van invoer- en uitvoergrootheden na). Een getal van het type FIXED DECIMAL(6,2) heeft een fixed-point representatie met zes decimale cijfers, waarvan twee rechts van de decimale punt. Een getal van het type FLOAT BINARY(31) heeft een floating-point representatie met 31 binaire cijfers. Als we uitgaan van onbegrensd grote precisie, dan heeft PL/I een oneindig aantal numerieke typen. Bij elke letterlijke getalsconstante hoort precies één numeriek type, dat uitsluitend uit de vorm ervan kan worden bepaald. De letterlijke constante '1234.56' is een FIXED DECIMAL (6,2), omdat er zes cijfers in voorkomen, waarvan twee rechts van de decimale punt. De letterlijke constante '0000101B' is een FIXED BINARY(7) en '0012 34E-3' is een FLOAT DECIMAL(6). Numerieke operaties kunnen alleen op operanden van gelijk type worden uitgevoerd. PL/I heeft automatische conversies van elk getalstype naar elk ander getalstype. In het manual van PL/I zijn tabellen te vinden die voor elke operatie aangeven welke conversies er plaatsvinden en wat het type van het resultaat is. Gewoonlijk is de PL/I-programmeur zich niet bewust van deze automatische conversies, waarvan er sommige tijdens het compileren plaatsvinden en andere tijdens het uitvoeren van het programma. In enkele gevallen geven de conversies verrassende resultaten (zie paragraaf 5.5).

Het attribuut SIZE geeft aan hoeveel cijfers precisie er in de representatie worden gebruikt. Dit attribuut geeft PL/I in beginsel de mogelijkheid tot machine-onafhankelijkheid. Maar in de praktijk gebruiken de meeste PL/I-compilers het attribuut SIZE om te bepalen welke van de verschillende machine-afhankelijke waarden gebruikt moet worden. Daardoor kan het gebeuren dat, als de programmeur FIXED BINARY(10) heeft gespecificeerd, de compiler in feite een veld van 16 bits gebruikt. Overflow buiten de tien bits zou dan niet worden ontdekt, tenzij de SIZE-conditie aangezet is. Maar programmeurs zetten de SIZE-conditie zelden aan, omdat het uitvoeren van het programma dan langer duurt.

Getallen in Ada

Er worden nog steeds nieuwe manieren bedacht om numerieke typen te organiseren. Ada kent maar twee numerieke typen, *universal_integer* en *universal_real* genaamd. In tegenstelling tot de traditionele getalstypen zijn deze twee typen werkelijk de gehele getallen en de reële getallen. Maar geen van beide typen is in Ada echt geïmplementeerd. De programmeur kan alleen *subtypen* van deze universele typen gebruiken. Een integer subtype wordt gespecificeerd met behulp van de onder- en de bovengrens van een bereik van gehele getallen. Pascal was de eerste taal waarin integer typen werden gespecificeerd met behulp van subtypen. Een dergelijke specificatie geeft meer informatie over het type van het getal, omdat die expliciet de verwachte onder- en bovengrens van de integers aangeeft.

Ada heeft enkele voorgedefinieerde, machine-onafhankelijke integer subtypen, namelijk `INTEGER`, `SHORT_INTEGER` en `LONG_INTEGER`, waarvan het bereik symmetrisch rond nul ligt. Het bereik van deze voorgedefinieerde subtypen kan worden bepaald aan de hand van de *attributen* `FIRST` en `LAST`, die de onder- en de bovengrens aangeven. De definitie van de taal Ada zegt expliciet dat de feitelijke implementatie van elk integer subtype één van de voorgedefinieerde typen is, het zogenaamde *oudertype* (parent type). De informatie over het bereik wordt gebruikt om vast te stellen welk van de voorgedefinieerde typen als oudertype zal worden gebruikt.

De informatie over het bereik wordt ook gebruikt als een beperking voor integers van het subtype en elke overtreding van die beperking veroorzaakt de exceptie `CONSTRAINT_ERROR`. Er zijn in Ada geen letterlijke constanten voor het aanduiden van de subtypen; alle letterlijke integer constanten zijn van het type *universal_integer*. Ada zorgt voor impliciete conversie van de universele getalstypen naar elk subtype.

De reals in Ada hebben een soortgelijke verzameling eigenschappen, maar er zijn extra complicaties. Er zijn twee soorten real typen: floating-point en fixed-point. Een floating-point real wordt gespecificeerd met behulp van het aantal decimale cijfers en eventueel nog met een beperking van het bereik (bestaande uit een onder- en een bovengrens). Een *fixed-point* real wordt gespecificeerd met behulp van een *delta* en een beperking van het bereik. Een delta specificeert de maximale tussenruimte tussen twee opeenvolgende getallen van dit type. Elk type definieert een verzameling getallen die de *modelgetallen* (model numbers) worden genoemd. Een implementatie van een type moet in staat zijn

de modelgetallen van dat type exact te representeren. Voor het uitleggen van real typen introduceert de definitie van Ada nog een ander geheimzinnig type, namelijk *universal_fixed*, maar dat wordt uitsluitend voor didactische doeleinden gebruikt.

Eén numeriek type?

Waarom moet een programmeur zich over al deze getalstypen zorgen maken om tot programmeren te komen? Een getal is tenslotte gewoon een getal. En er zijn inderdaad talen die getallen gewoon als getallen behandelen. Zowel APL als SNOBOL hebben één numeriek type en de programmeur denkt er niet over na welk soort getalsrepresentatie hij moet gebruiken. De programmeertaal neemt de zorg voor dat soort details over. Waarom hebben dan niet alle talen maar één numeriek type? Het probleem zit in de efficiëntie. Om getallen efficiënt te kunnen implementeren heeft de programmeertaal enige hulp nodig. De meeste machines beschikken over minstens twee verschillende manieren om getallen voor te stellen. In programmeertalen wordt dat verschil weerspiegeld in het gebruik van verschillende numerieke typen.

Operaties op getallen

De klassieke operaties optellen, aftrekken, vermenigvuldigen en delen geven problemen als ze worden toegepast op deelbereiken van de gehele getallen. De optelling en de aftrekking zijn volledige functies op de verzameling van alle getallen, maar zijn partiële functies op de deelbereiken van de gehele getallen. Omdat met representaties op computers slechts een deelverzameling van de gehele getallen wordt gerepresenteerd kunnen alle vier de operaties overflow of underflow veroorzaken. Dit probleem heeft taalontwerpers en programmeurs steeds dwars gezeten. Taalontwerpers kunnen talen ontwerpen zonder beperkingen op de integers. Programmeurs kunnen elegante programma's schrijven door ervan uit te gaan dat er geen grenzen zijn gesteld aan de integers. Maar om realistisch te blijven moeten zowel taalontwerpers als programmeurs rekening houden met problemen als overflow.

Bij floating-point getallen treedt overflow op een andere manier op. Ten eerste kan de exponent buiten de gestelde grenzen komen. Maar het feit dat de resultaten van operaties benaderingen zijn, vormt een ernstiger probleem. Als het resultaat van een operatie meer cijfers bevat dan in de toegemeten hoeveelheid

geheugen passen, dan worden de minst significante cijfers die te veel zijn, weggegooid. Daardoor zijn de floating-point operaties niet altijd exact. Integer operaties zijn altijd exact als ze geen overflow veroorzaken. Het feit dat floating-point getallen maar benaderingen zijn, kan tot resultaten leiden die als een verrassing komen voor wie niet op de hoogte is met de manier waarop computers rekenen.

Zo is een competente programmeur er niet verbaasd over dat 3*(1/3) niet gelijk is aan 1. De uitkomst ligt heel dicht bij 1, is misschien .9999999, maar is niet gelijk aan 1. Daarom is het een slecht gewoonte om in een programma te vragen of twee floating-point getallen precies gelijk zijn. Het is redelijk om te vragen of het ene getal groter is dan het andere, maar onredelijk om te vragen of ze gelijk zijn. In de implementatie van APL werd een nieuwe aanpak van dit verschijnsel geïntroduceerd. Falkoff en Iverson (1987) schrijven:

> "Eén van de problemen die we niet hadden zien aankomen was het leveren van zinnige resultaten bij het vergelijken van grootheden die maar met beperkte precisie gerepresenteerd waren. Als bijvoorbeeld de waarde van `X` en `Y` gegeven werden door `Y←2÷3` en `X←3 × Y`, dan wilden we graag dat de vergelijking `2=X` de waarde `1` (die voor true staat) zou opleveren, zelfs als de representatie voor de grootheid `X` iets van `2` zou afwijken.
>
> Dit werd opgelost door de introductie van een tolerantie bij vergelijkingen (door L.M. Breed *fuzz* gedoopt ...). Die tolerantie werd vermenigvuldigd met het grootste van de twee argumenten en dat leverde de tolerantie die bij de betreffende vergelijking moest worden toegepast. De tolerantie was eerst vast (gelijk aan `1E-13`), maar later kon de gebruiker de waarde zelf opgeven. De zaak is moeilijker gebleken dan we aanvankelijk dachten en de discussie erover duurt nog steeds voort..."

Opgaven

1. Los het volgende probleem op voor de hypothetische machine uit paragraaf 2.4 en gebruik daarbij de schaalfactorenmethode. Er zijn drie invoergrootheden: A ligt in het bereik 4000–500 000, B in het bereik .0001 – 100 en C in het bereik 1000–5000. De uiteindelijke uitvoer (in miljoenen) wordt gespecificeerd door 100 * A + C ** 2 + B * C ** 3.

2. Bespreek de voor- en nadelen van het berekenen van de exponent bij fixed-point en floating-point getalstypen tijdens het uitvoeren van het programma dan wel tijdens het compileren. Bij fixed-point moet de compile-time type-controle gebeuren op basis van de exponenten, terwijl bij floating-point al dit werk (het berekenen van de exponenten) tijdens het uitvoeren van het programma gebeurt.

3. Beschouw de volgende representatie voor getallen. Een rationaal getal is de verhouding van twee integers. Om zulke getallen te representeren kunnen we met twee integers volstaan. Vermenigvuldigen en delen gaat eenvoudig, omdat daarbij alleen het vermenigvuldigen en delen van integers nodig is. De optelling en de aftrekking zijn ingewikkelder, maar die kunnen ook worden geïmplementeerd met behulp van de vier basis-operaties op integers. Zo is $(a/b) + (c/d) = (ad + cb)/bd$. Rationale getallen hebben een aantal grote voordelen boven floatingpoint representaties. Gewone rationale getallen, zoals 1/3 en 1/7 kunnen er exact mee worden gerepresenteerd. Als de expressie (3*(1/3)) wordt uitgewerkt, is het resultaat 1, niet .99999999. Er behoeven geen ingewikkelde floating-point operaties te worden geïmplementeerd. De rationale operatoren zijn gemakkelijk op te bouwen met behulp van de integer operaties. Wat zijn dan de nadelen van rationale representaties?

4. Op welke problemen zouden Falkoff en Iverson gedoeld hebben bij het behandelen van de fuzz factor (aan het eind van paragraaf 2.4)?

3

Samengestelde typen

Een samengesteld type is een gegevenstype waarvan de waarden collecties van gegevens zijn. Dit in tegenstelling tot de basistypen uit het vorige hoofdstuk, waarvan de waarden ondeelbare gegevenselementen zijn. Een samengesteld type *definieert* geen nieuw basistype, maar *construeert* een nieuw gegevenstype uit reeds bestaande gegevenstypen. Om die reden worden samengestelde typen soms constructortypen genoemd. Gewoonlijk vindt de toegang tot elk element van een samengestelde waarde plaats via een of andere *indexwaarde*. De organisatie en implementatie van de collectie bepaalt in hoge mate de toegangsmethode. In dit hoofdstuk onderzoeken we drie belangrijke samengestelde typen: arrays, records en sequences. Arrays vormen het meest voorkomende samengestelde type. Een array wordt (in dit boek) gekarakteriseerd als een collectie van vaste omvang, waarbij elk element dynamisch toegankelijk is. Een record kan ook worden gekarakteriseerd als een collectie van vaste omvang, maar de elementen zijn op een beperktere manier toegankelijk dan in een array. Een sequence is een collectie waarvan de omvang tijdens de uitvoering van het programma dynamisch veranderd kan worden.

3.1 Arrays

Net als getallen behoren arrays tot de oudste typen. Al voor de opkomst van programmeertalen werden in assembleertalen arrays gebruikt. In de wiskunde en de exacte wetenschappen zijn variabelen met een onder- (of boven-)index een gebruikelijke notatie. Variabelen die van een index worden voorzien, dui-

den een collectie van gegevens aan. Een index wijst dan een bepaald element van de collectie aan. De index kan worden gebruikt om een tijdstip aan te duiden, of een plaats of een positie in een of andere volgorde waarin de gegevens kunnen zijn gesorteerd. Geïndexeerde variabelen worden in het geheugen gerepresenteerd als een aaneensluitend stuk geheugen, met de index als toegang.

In de begintijd beschouwden taalontwerpers arrays niet als een gegevenstype. Het klassensysteem dat in vliegtuigen wordt gehanteerd, kan als voorbeeld worden gebruikt om te beschrijven in welke mate gegevenstypen gelijk worden behandeld. *Eerste-klassepassagiers* hebben alle rechten, waartoe in het algemeen behoren toekenning, letterlijke constanten, vergelijkingsoperatoren en de mogelijkheid als parameter te worden doorgegeven. Twee typen behoren tot dezelfde klasse als alles wat met het ene type kan worden gedaan, ook met het andere type kan worden gedaan. In bijna alle talen hebben de getalstypen de meeste rechten. Andere typen hebben dan een kleiner scala van mogelijkheden en worden dus als *tweede-klassepassagiers* beschouwd. In de meeste talen uit de begintijd, waaronder FORTRAN en ALGOL 60, horen arrays niet in de eerste klasse thuis. In geen van beide talen kan een array-waarde worden toegekend of met een andere vergeleken. De individuele elementen van een array kunnen wel worden toegekend of vergeleken, maar de gehele collectie niet. In FORTRAN zijn arrays geordende verzamelingen gegevens waarnaar met een symbolische naam wordt verwezen; ze vormen geen gegevenstype. Evenzo is een array in ALGOL 60 een collectie van *fictieve variabelen*, in plaats van een variabele met als waarde een collectie waarden. Zulke uitgangspunten leiden meestal tot ingewikkelde taaldefinities. De beschrijving van het doorgeven van parameters kan bijvoorbeeld onevenredig ingewikkeld worden, omdat gegevenstypen en arrays apart moeten worden beschreven.

PL/I en APL behoorden tot de eerste talen die arrays bijna alle rechten van eerste-klassepassagiers gaven. PL/I staat het toekennen van arrays toe en bevat uitbreidingen van vele rekenkundige operatoren tot arrays van getalstypen. Maar PL/I kent toch nog enkele beperkingen, zoals het feit dat procedures geen arrays als waarde kunnen afleveren. Noch PL/I noch APL staat arrays van arrays toe. Hoewel arrays in APL dynamisch van lengte kunnen veranderen, worden ze toch in dit hoofdstuk behandeld, omdat APL een belangrijke bron is van ideeën over arrays. In ALGOL 68 en Pascal kregen arrays zelfs nog meer rechten, en ondanks enkele kleine beperkingen zijn het daar eerste-klassepassagiers geworden. In beide talen vormen arrays een gegevenstype; er hoort een verzameling waarden en een verzameling operaties bij.

We zullen vele moeilijke en subtiele kenmerken van arrays bespreken. We onderzoeken arrays hier meer aan de hand van hun kenmerken dan per taal. De eerste twee kenmerken van een array zijn het indextype en het elementtype. Het indextype is het type van de waarde die wordt gebruikt om een element van een array te selecteren en het elementtype is het type van de elementen die in de array zijn opgeborgen.

Indextypen

Een *index* selecteert een element van een array. De index heeft een type, het *indextype*. Indextypen zijn bijna altijd integers. Het gebruik van integers als indextype van een array is zo algemeen en natuurlijk, dat vele programmeertalen alleen integers als indextype toestaan. Opdat een array efficiënt in het geheugen kan worden opgeborgen moet de array *grenzen* hebben. De grenzen geven aan welke waarden geldige indices zijn. De grenzen worden meestal gespecificeerd met behulp van een onder- en een bovengrens. De ondergrens geeft de kleinste waarde aan die als een geldige index kan worden gebruikt en de bovengrens de grootste waarde die als een geldige index kan worden gebruikt. In de meeste toepassingen is de ondergrens 0 of 1. Dit gebeurt zo vaak, dat sommige talen van de programmeur alleen een opgave van de bovengrens verlangen. Sommige talen staan de programmeur niet eens toe de ondergrens op te geven. FORTRAN hanteert 1 als ondergrens en sommige dialecten van BASIC hebben 0 als ondergrens; de programmeur kan daarin geen verandering aanbrengen. Te beginnen met Pascal hebben vele programmeertalen ook opgesomde typen als indextype toegelaten. Die mogelijkheid biedt een geschikte vorm van documentatie en helpt fouten te voorkomen. Neem als voorbeeld de volgende tweedimensionale array die geïndexeerd wordt met een kleur en een object:

```
type KLEUR is (ROOD, BLAUW, GEEL, WIT);
type OBJECT is (BOL, KUBUS, CILINDER);
type MATRIX is array (KLEUR, OBJECT) of INTEGER;
V: MATRIX;
begin
        V(BLAUW, KUBUS) := 5;
        ...
```

Behalve dat dit programmafragment goed leesbaar is, is het ook onmogelijk de beide indices door elkaar te halen. Als de programmeur zou denken dat de kleur de tweede index was en `V(KUBUS, BLAUW)` had geschreven, zou de compiler hem op de fout wijzen. Opgesomde typen als indextypen zijn even gemakkelijk te

implementeren als integer typen, omdat de meeste compilers opgesomde typen als integers representeren. Andere indextypen zijn vaak veel ingewikkelder te implementeren. Zo is in SNOBOL de implementatie van tabellen, die strings als indextype hebben, veel ingewikkelder dan die van de arrays van SNOBOL.

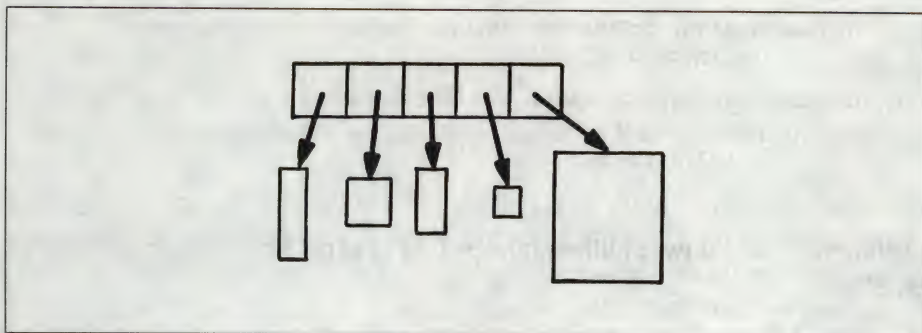
Als we afzien van de implementatie, kunnen we gewoon zeggen dat een array een waarde van het elementtype toevoegt aan elke waarde van het indextype. Deze generalisatie van een array wordt een *afbeelding* genoemd. Elk type kan fungeren als indextype van een afbeelding.

Bij meerdimensionale arrays heeft elke index zijn eigen indextype. Tabellen in SNOBOL zijn afbeeldingen met strings als indices. Een afbeelding is een functie van de waarde van het indextype naar een waarde van het elementtype. Sommige hogere programmeertalen, zoals SETL, staan willekeurige afbeeldingen toe. Vele van de punten die hierna worden besproken, kunnen even goed op afbeeldingen worden toegepast als op arrays, maar de meeste programmeertalen staan geen willekeurige afbeeldingen toe, omdat niet bekend is hoe die efficiënt kunnen worden geïmplementeerd. Omdat het begrip array is ontstaan uit een bepaald soort implementatie en de bijbehorende methode van opslag in het geheugen, is het niet redelijk afbeeldingen met een ingewikkeld indextype als arrays te beschouwen.

Elementtypen

Het elementtype is het type van de elementen die in de array zijn opgeslagen. Gewoonlijk is het elementtype een getalstype. Maar in het algemene geval kan elk type als elementtype fungeren. PL/I kent arrays van elk type, maar het elementtype mag zelf niet weer een array zijn. Maar PL/I staat wel arrays van structuren van arrays toe! ALGOL 68 en Pascal leggen geen beperkingen op aan elementtypen. Om efficiënte implementatie van arrays mogelijk te maken, moeten de waarden van een elementtype een vaste hoeveelheid geheugen beslaan. Omdat zowel in Pascal als in ALGOL 68 elk type een vaste hoeveelheid geheugen in beslag neemt, kunnen in die talen alle arrays op een efficiënte manier worden geïmplementeerd.

Een *homogene* array is een array met slechts één elementtype. Met andere woorden, alle elementen van een homogene array zijn van hetzelfde type. Een *heterogene* array is een array waarvan de elementen van verschillende typen kunnen zijn. De meeste traditionele programmeertalen staan geen heterogene



Figuur 3-1 Een niveau van indirectie zoals gebruikt in arrays in SNOBOL.

arrays toe, omdat die onverenigbaar zijn met het beginsel van controle tijdens het compileren en omdat ze moeilijker te implementeren zijn. Maar SNOBOL heeft wel heterogene arrays, die geïmplementeerd worden als arrays van pointers naar waarden. Door die opzet kan in SNOBOL elk element van een array een bepaalde vaste grootte hebben, en daardoor kan de toegang tot de elementen van een array efficiënt zijn, ook al zijn er elementen van verschillende grootte in het spel. De echte kosten van arrays in SNOBOL zitten in het geheugenbeheer. Figuur 3-1 laat een array in SNOBOL zien met waarden van vele verschillende afmetingen.

Dimensies en nesting van arrays

Een array kan één of meer dimensies hebben. Elke dimensie heeft zijn eigen type en zijn eigen grenzen. De omvang van een dimensie is het aantal geldige indexwaarden in die dimensie; voor een integer index is de omvang eenvoudig 1 meer dan het verschil tussen boven- en ondergrens. Het totaal aantal elementen van een array is het produkt van de omvang van alle dimensies. Voor de toegang tot een element van een array moet voor elke dimensie van de array een index worden opgegeven. Er kan een grens gesteld zijn aan het aantal dimensies van een array, maar in de praktijk wordt die zelden bereikt, omdat de overgrote meerderheid van arrays minder dan een zestal dimensies heeft. Zelfs als een taal arrays met meer dan één dimensie niet toestaat, is het mogelijk toch meerdimensionale arrays te gebruiken door van een functie gebruik te maken die een willekeurig aantal indices omrekent tot één index.

We kunnen bijvoorbeeld een matrix van vijf bij vijf elementen krijgen door op de volgende manier een vector van 25 elementen en een functie te gebruiken:

```
F: function (X, Y:INTEGER) return INTEGER is
        begin return (X-1)*5+Y end

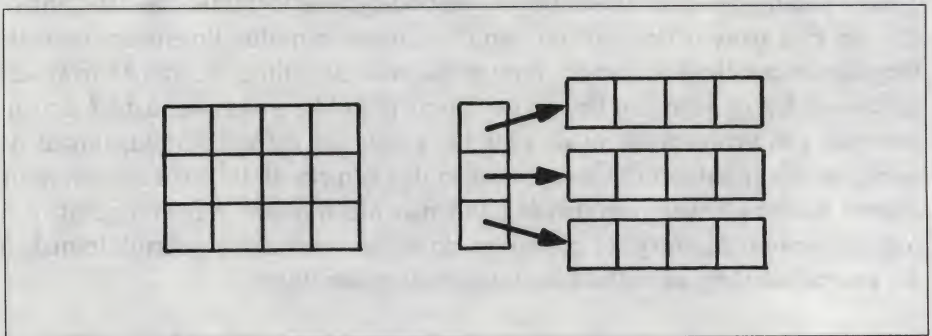
V: array (1..25) of INTEGER;
begin
        V(F(3, 4)) := 5;
        ...
```

De functie F beeldt twee indices (tussen 1 en 5) af op één enkele index tussen 1 en 25.

Een andere manier om meerdimensionale arrays tot stand te brengen is het nesten van arrays. Een ééndimensionale array van ééndimensionale arrays kan als een tweedimensionale array worden gebruikt, alleen de syntaxis is verschillend:

```
V: array (1..5) of array (1..5) of INTEGER;
begin
        V(3)(4) := 5;
        ...
```

In dit voorbeeld is V een array van arrays. We kunnen een element van één van de binnenste arrays krijgen door twee indices te gebruiken die gescheiden worden door ')(' in plaats van de gebruikelijke komma. Figuur 3-2 laat het verschil in opzet zien tussen een tweedimensionale array en een ééndimensionale array van ééndimensionale arrays. Men zou zich kunnen afvragen wat de wezenlijke verschillen zijn tussen deze varianten. Want in zekere zin zijn ze allemaal gelijk, omdat ze allemaal 25 elementen hebben en de toegang geschiedt via twee indices tussen 1 en 5. ALGOL 68 en Pascal demonstreren enkele van de ver-



Figuur 3-2 Een tweedimensionale array en een ééndimensionale array van ééndimensionale arrays.

schillen. Pascal kent strikt genomen alleen ééndimensionale arrays en beschouwt een meerdimensionale array als een verkorte schrijfwijze voor een geneste array. De volgende twee typen hebben in Pascal dan ook dezelfde betekenis:

```
type EEN is array (1..5, 1..5) of INTEGER;
type TWEE is array (1..5) of array (1..5) of INTEGER;
```

In ALGOL 68 zijn dit verschillende typen. ALGOL 68 staat bijvoorbeeld verschillende grenzen toe voor elk van de ééndimensionale arrays, hetgeen inhoudt dat voor elke ééndimensionale array een verschillende hoeveelheid geheugen wordt gebruikt en dat de ééndimensionale versie via een ander mechanisme in het geheugen wordt opgeslagen dan de tweedimensionale array. In ALGOL 68 is het ook toegestaan operaties toe te passen op elk van de ééndimensionale arrays als geheel. Pascal staat dergelijke operaties niet toe, evenmin als verschillende grenzen.

Arraywaarden

Als een gegevenstype een verzameling waarden is, wat is dan de verzameling waarden van een arraytype? In de wiskunde komen arraywaarden overeen met zaken als vectoren, matrices en Cartesische produkten, maar in al deze wiskundige objecten speelt het begrip index geen rol. De grenzen moeten ergens in het spel worden betrokken, maar er valt over te twisten waar dat precies moet gebeuren. ALGOL 68 en Pascal leveren ook hier weer klassieke voorbeelden van het verschil. In ALGOL 68 vormen de grenzen een deel van de waarde van de array, niet van het type. In Pascal zijn de grenzen een deel van het type van de array, niet van de waarde. Dit verschil heeft enorme gevolgen voor de manieren waarop arrays kunnen worden gebruikt.

De typen

```
array (1..3) of INTEGER
```

en

```
array (5..7) of INTEGER
```

zijn in Pascal twee verschillende typen. De waarden van deze beide typen zijn toevallig gelijk, namelijk de verzameling van alle vectoren van drie integers.

Omdat in Pascal het type van een argument moet overeenkomen met het type van de parameter, is het onmogelijk in Pascal procedures te schrijven die op arrays met verschillende grenzen werken. In enkele recente Pascal-standaards is dit veranderd; daarin is een *conformant-array-schema** mogelijk, dat array-parameters met algemene grenzen toestaat.

Het type

```
array () of integer
```

is een type in ALGOL 68.** De waarden van dit type zijn alle ééndimensionale arrays van integers. Elke arraywaarde in ALGOL 68 bestaat uit de grenzen plus de elementen van de array. Deze verzameling bevat de arrays met drie elementen (3,4,5) met grenzen 1 en 3, en (3,4,5) met grenzen 5 en 7. Deze beide arraywaarden zijn weliswaar verschillend, maar behoren tot dit ene type. De toekenning van arrays is ingewikkeld in ALGOL 68. Alleen waarden met dezelfde grenzen kunnen aan een arrayvariabele worden toegekend. Als de grenzen van de toe te kennen arraywaarde verschillen van die van de arrayvariabele, dan kan de ondergrens *herzien* worden met het teken @. Bij arrayparameters hoeven in ALGOL 68 geen grenzen te worden gespecificeerd. In plaats daarvan worden arrays met hun grenzen doorgegeven. Deze opzet verschaft een natuurlijke manier om procedures te schrijven voor arrays met verschillende grenzen.

APL heeft weer een andere interessante aanpak van arrays. Het begrip *vorm* (shape) geeft aan hoe de elementen van een vector of een matrix geordend moeten worden. De APL-expressie

```
A ← 1 4 9 16 25 36
```

kent aan de variabele A een vector van zes elementen toe. De variabele A kan nu in de *vorm* worden gebracht van een matrix met twee kolommen en drie rijen met behulp van de expressie

```
A δ 2 3
```

Het blijkt dus dat in APL een array in wezen steeds een ééndimensionale array is, die met elk gewenst aantal dimensies kan worden geïnterpreteerd.

---

* Addyman (1980)

** Technisch correct uitgedrukt: *row of integral* is een mode in ALGOL 68.

Arraygrootte en tijdstip van allocatie

De grenzen van een array bepalen de grootte van de array en de hoeveelheid geheugen die nodig is om de array te representeren. Dat geheugen moet vóór het gebruik worden *gealloceerd*, dat wil zeggen gezocht en toegewezen. Het moment waarop de allocatie plaatsvindt heeft grote invloed op de flexibiliteit van de arraygrenzen. In sommige talen, zoals FORTRAN en Pascal, vindt de allocatie tijdens het compileren plaats. Daartoe moeten de feitelijke grenzen tijdens het compileren aan de compiler bekend zijn. Andere talen alloceren arrays pas tijdens het uitvoeren van het programma. Na de allocatie zijn de grenzen en de grootte van de array onveranderbaar. In vele toepassingen is het wenselijk dat arrays tijdens het uitvoeren van het programma op de juiste grootte worden gealloceerd. Run-time allocatie heeft twee voordelen boven allocatie tijdens het compileren. Omdat de programmeur arrays op precies de juiste grootte kan alloceren, wordt er geen geheugen verspild. En de programmeur hoeft niet tevoren de grootste omvang te bepalen die voor de array nodig kan zijn. Het veranderen van de grenzen ná de allocatie vereist een radikaal verschillend opslagmechanisme; het daarvoor nodige type noemen we een *sequence*.

Operaties op arrays

De meest fundamentele operatie op arrays is indexering. De operanden zijn de array en de indices, en de afgeleverde waarde is een element van de array. De syntaxis van de operatie ziet er in het algemeen uit als die van een aanroep van een functie, maar sommige talen gebruiken vierkante haken in plaats van ronde. In de meeste talen kan er aan het gekozen array-element ook een waarde worden toegekend. De indexeringsoperatie levert een fout op als één van de indices buiten de grenzen valt. Dit is, afgezien van enkele gespecialiseerde invoer/uitvoer-operaties, de enige manier om in talen zoals FORTRAN en ALGOL 60 arrays te gebruiken. Die beperktheid komt voort uit het feit dat arrays tweedeklassepassagiers zijn.

APL is vooral bekend om zijn vector- en matrixoperaties. APL beschikt over een groot scala van operaties die arrays creëren, transformeren, combineren en manipuleren. In APL zijn expressies in feite array-expressies in plaats van numerieke expressies. PL/I bevat een beperkte verzameling van deze array-operaties, waaronder toekenning, vergelijking en componentsgewijze uitgevoerde numerieke operaties. Men kan in PL/I subarrays selecteren. Als in APL A en B twee arrays van 5 bij 5 integers zijn, dan levert de expressie

```
A(*,3) + B(2,*)
```

als waarde de ééndimensionale array die ontstaat als de derde kolom van A component voor component bij de tweede rij van B wordt opgeteld. In ALGOL 68 is deze mogelijkheid uitgebreid, doordat bij het selecteren van subarrays ook grenzen zijn toegestaan.

Als A en B in ALGOL 68 twee arrays zijn van 5 bij 5 integers, dan levert de expressie

```
A(1:3, 3) + B(2, 3:5)
```

als waarde de ééndimensionale array die ontstaat als de eerste drie elementen van de derde kolom van A worden opgeteld bij de laatste drie elementen van de tweede rij van B. In ALGOL 68 is een *trimscript* ofwel een trimmer ofwel een index. Een *trimmer* is in ALGOL 68 een paar getallen die de onder- en bovengrens aangeven, zoals in `1:3`. Een trimmer selecteert, net als een index, een deel van een array. Maar terwijl een index een enkel element selecteert, selecteert een trimmer een reeks elementen, vanaf de ondergrens tot en met de bovengrens van de trimmer. Merk op dat het type van `A(1,3)` integer is, maar het type van `A(1:3,3)` array van integer.

In APL telt de unaire operator `+/` de elementen van een array bij elkaar op. Als de array `A` bijvoorbeeld gelijk is aan `(3,4,5)`, dan levert `+/A` de waarde `12` af. APL heeft verscheidene operatoren van de vorm

```
□/
```

waarin □ één of andere operator is. Elke operator van deze vorm werkt op dezelfde manier. De expressie `*/A` levert het produkt van de elementen van `A` als waarde af. In APL kan deze klasse van operatoren niet worden uitgebreid, maar men kan zich een taal voorstellen die dit idee generaliseert. Een *meta-operator* is een operator op operatoren en wordt ook wel *functionaal* genoemd, omdat de operanden functies zijn.

De definitie van `/` zou kunnen zijn (□ stelt een operator met twee operanden voor):

```
□/ (a1, a2, ..., an) = a1 □a2 □... □ an
```

De Functional Programming (FP) language, ontworpen door John Backus (1978b) heeft vele meta-operatoren, waaronder `/`. In sommige programmeertalen kan de programmeur nieuwe meta-operatoren definiëren.

Een ander nuttige meta-operator voor arrays is de operator "pas toe op elk element", die een unaire operator als operand heeft en deze op alle elementen van een array element voor element toepast; het resultaat is een array met dezelfde omvang en dezelfde dimensies. Deze operator kan gedefinieerd worden door:

```
α □ (a1, a2, ..., an) = ( □ a1, □ a2 , ..., □an)
```

Deze operator α kan worden gegeneraliseerd tot binaire en *n*-aire operaties. In sommige talen komen wel bepaalde van deze operatoren voor; zo zijn in PL/I vele van de numerieke operatoren gegeneraliseerd, zodat ze ook op arrays werken. Maar in andere talen, met name FP, kan elke operator zo worden gebruikt.

Diverse punten

Een array is een collectie van gegevens. Hoe klein kan die collectie worden? Een array van twee elementen zal in elke taal wel toegestaan zijn; maar hoe zit het met één of nul elementen? Men zou in eerste instantie kunnen denken dat arrays met één of nul elementen onzin zijn en alleen om wille van de volledigheid toegestaan zouden moeten worden. Maar een praktische reden doet zich bijvoorbeeld voor in een programma dat een array alloceert waarvan de omvang afhangt van het aantal invoergegevens. Alleen als er extra code is geschreven voor speciale gevallen, zou zo'n programma correct werken voor minder dan twee invoergegevens. Talen als FORTRAN, PL/I en Pascal staan wel arrays met één element toe, maar niet met nul elementen. Andere talen, zoals ALGOL 68, Ada en APL staan zowel arrays met nul als met één element toe. Er zijn geen talen die nul elementen wel en één element niet toestaan; maar ALGOL 68, waarin wel array-constanten voor arrays met nul of twee elementen mogelijk zijn, kent er geen voor arrays met één element!

Het feit dat een taal geen constanten kent voor arrays laat weer zien dat dit type een tweede-klassepassagier is. DATA-opdrachten in FORTRAN en initialisatie-attributen in PL/I zijn voorbeelden van beperkte manieren om een constante array aan te geven. APL was een van de eerste talen waarin arrayconstanten toegestaan werden. De elementen van een array worden in APL gescheiden door spaties (bijvoorbeeld `3 4 5`). In ALGOL 68 kan een array worden weerge-

geven als een lijst waarden die door komma's worden gescheiden (bijvoorbeeld `(3,4,5)`). In talen waarin dit soort expressie ontbreekt, moet men om hetzelfde resultaat te bereiken, zijn toevlucht nemen tot *for*-lussen of series toekenningsopdrachten. De arrayconstanten in APL en ALGOL 68 kunnen beter arrayconstructors worden genoemd, omdat elk element een willekeurige expressie kan zijn in plaats van een eenvoudige constante. In ALGOL 68 construeert de expressie `(X, Y, Z+1)` een array met drie elementen, waarvan de waarde bij elke evaluatie verschillend kan zijn, omdat de variabelen een verschillende waarde kunnen hebben.

3.2 Records

COBOL was de taal die het type record heeft geïntroduceerd. Een record-type is, net als een arraytype, een collectie waarden die elk afzonderlijk kunnen worden benaderd. De elementen van een record worden *velden* genoemd. Anders dan bij arrays mogen de verschillende velden van een record van verschillend type zijn. De verzameling waarden die door een recordtype worden gerepresenteerd is het Cartesisch produkt van de typen van de velden.

De operatie die toegang biedt tot een veld van een record heet *selectie*. De velden van een record worden aangeduid met een identifier, die de *selector* wordt genoemd. Selectie en selectors zijn equivalent met indexering en indices bij arrays. Maar een belangrijk verschil is dat een index een willekeurige expressie mag zijn, die tijdens het uitvoeren van het programma bij elke evaluatie een verschillende waarde kan opleveren. Records hebben geen indextype en selectors zijn geen gegevenstype. Een selector kan dan ook niet worden vervangen door een variabele of een expressie. Dat is belangrijk voor talen waarbij de typen tijdens het compileren worden gecontroleerd; een compiler zou anders niet altijd kunnen vaststellen wat het type is van het resultaat van een selectie. Talen met run-time typecontrole hebben geen behoefte aan deze beperking. Maar zulke talen zouden zowel voor traditionele arrays als voor records gewoon heterogene arrays kunnen gebruiken.

Declaratie, initialisatie en selectie van records geschiedt in verschillende talen op verschillende manieren. Twee voorbeelden illustreren enkele gebruikelijke methoden.

In de traditie van ALGOL en Pascal krijgt ieder veld een unieke naam, zoals:

```
type PERSOON is
     record
          NAAM:        STRING;
          LEEFTIJD:    INTEGER;
          GEWICHT:     REAL;
     end record;

X,Y: PERSOON;
```

De variabelen X en Y zijn records die bestaan uit drie velden, genaamd NAAM, LEEFTIJD en GEWICHT. In ALGOL 68 wordt de waarde van een record (net als van een array) weergegeven als een lijst waarden tussen haakjes, met één waarde per veld van het record. Op de volgende manier kan aan de variabele X een waarde worden toegekend:

```
X := ("Jan", 4, 18.1);
Y.LEEFTIJD := X.LEEFTIJD;
```

In de tweede opdracht komt selectie voor. De meeste talen gebruiken voor de selectie van een veld als syntaxis:

```
X.LEEFTIJD
```

Dit kan worden beschouwd als een operator waarvan het ene operand de naam van een selector is en het andere een record. ALGOL 68 gebruikt de notatie

```
LEEFTIJD of X
```

Het tweede voorbeeld van records dat we hier beschrijven is de aanpak van SNOBOL. Het volgende fragment is de SNOBOL-versie van het vorige voorbeeld:

```
DATA('PERSOON(NAAM, LEEFTIJD, GEWICHT)')
X = PERSOON('JAN', 4, 18.1)
Y = PERSOON()
LEEFTIJD(Y) = LEEFTIJD(X)
```

In SNOBOL zijn creatie en selectie functies. De functie DATA roept functies in het leven voor elke identifier in de parameter, n om het record te construeren en n voor elk veld. Omdat in SNOBOL variabelen geen type hebben, wordt aan X, Y, NAAM, LEEFTIJD en GEWICHT geen type gegeven.

Afhankelijk van de taal kunnen de grenzen van een array al of niet deel uitmaken van het type. Op dezelfde manier kunnen de selectors van een record al

of niet deel van het type zijn. In ALGOL 68 zijn de grenzen van een array geen deel van het type, maar horen de selectors van een record wel bij het type. Andere talen (zoals PL/I) beschouwen de namen van de selectors niet als deel van het type. Daarom zou de laatste opdracht van het volgende programmafragment (op de juiste wijze vertaald) wel correct PL/I zijn, maar geen correct ALGOL 68.

```
type COMPLEX is
        record
                RE, IM: REAL;
        end record;

type COORDINATEN is
        record
                X, Y: REAL;
        end record;

WAARDE: COMPLEX;
PLAATS: COORDINATEN;

begin
        WAARDE := PLAATS;
        ...
```

De toekenning hierboven is ook in Ada verboden, maar om een andere reden. Ada gaat uit van naam-equivalentie (zie paragraaf 5.6) en de namen van de typen van `WAARDE` en `PLAATS` zijn verschillend.

3.3 Sequences en strings

Een *sequence* is een array waarvan de omvang gewijzigd kan worden gedurende de uitvoering van het programma. Voor het implementeren van sequences is een andere strategie nodig dan voor arrays. Als voor een array eenmaal geheugen is gealloceerd is er naderhand geen ander geheugen meer nodig. Maar een sequence kan slinken of groeien en vóór de allocatie is niet altijd bekend hoeveel geheugen er nodig zal zijn. Omdat arrays en sequences zoveel op elkaar lijken, maken sommige talen er geen verschil tussen. In ALGOL 68 is er maar een klein verschil. ALGOL 68 eist het sleutelwoord `flex` bij elke array die tijdens uitvoering van het programma van omvang kan veranderen. Sommige talen hebben geen enkel soort sequence. Andere talen maken een uitzondering voor de nuttigste soort sequence, de *string* van tekens.

Een string is een sequence van tekens. In tegenstelling tot numerieke arrays van vaste omvang, die heel nuttig zijn, zijn strings van vaste lengte zeer onhandig in het gebruik. Strings worden gebruikt voor namen, adressen, etiketten, titels en allerlei zaken die geen vaste grootte hebben (behalve postcodes en drieletterwoorden). Het is interessant om de veranderingen te volgen die zich sinds de vroegste programmeertalen tot de hedendaagse talen hebben voorgedaan in de behandeling van strings. FORTRAN had geen strings, maar in latere versies mochten gegevens die uit tekens bestonden, bewaard worden in numerieke variabelen. FORTRAN stond ook strings toe in uitvoerformaten. In ALGOL 60 is het niet veel beter. Daarin mogen strings als parameters aan andere functies worden doorgegeven. PL/I kent zowel strings van vaste als van variabele lengte, maar eist bij strings van variabele lengte wel opgave van de maximale lengte. Het geheugenbeheer voor strings van variabele lengte in PL/I is eenvoudig; er wordt gewoon geheugen gealloceerd voor de maximale lengte van de string plus geheugen voor het veld waarin de lengte wordt bijgehouden. Maar voor de lengte van echte strings geldt geen beperking, die kunnen elke lengte hebben. Voor het implementeren van echte strings is dynamisch geheugenbeheer nodig. SNOBOL was, en is wellicht nog steeds, de beste beschikbare taal voor het manipuleren van strings. In SNOBOL zijn niet alleen echte strings beschikbaar, er is ook een breed scala van functies op strings, waaronder een zeer geavanceerd hulpmiddel voor pattern matching. Omdat voor echte strings een slim geheugenbeheer nodig is, beschikken vele talen (zelfs nieuwe) niet over strings. Pascal heeft alleen strings van vaste lengte, omdat bij het ontwerpen van Pascal efficiënte, eenvoudige implementatie één van de doeleinden was.

Eén van de fundamentele operaties op strings en sequences is *concatenatie* of samenvoeging. Concatenatie is een operatie op twee sequences. Het resultaat is een sequence waarvan het eerste deel bestaat uit het eerste operand en het tweede deel uit het tweede operand. Gebruikelijke operaties in veel talen zijn zoekoperaties, pattern matching en verificatie. Andere operaties die men vindt in talen die rijk genoeg zijn voorzien, zijn het selecteren van deel-sequences en het vervangen van de ene deel-sequence door een andere. Het vergelijken van strings is meestal gebaseerd op de lexicografische volgorde, dus die van een woordenboek. Dat betekent dat de string ‘CRAIG’ kleiner is dan ‘TINA’, ook al is ‘TINA’ korter dan ‘CRAIG’.

Strings vormen niet de enige nuttige soort sequence. In LISP zijn lijsten sequences van atomen en lijsten. De beroemde operaties van LISP op lijsten zijn

```
cons          car          cdr
```

Deze namen stammen van opdrachten in de machinetaal waarin de eerste implementatie van LISP werd geschreven. Andere talen en systemen geven deze operaties andere namen, zoals

```
makelist      first        rest
```

of misschien

```
appendleft    value        next
```

De eerste operatie construeert een langere lijst door met concatenatie een element voor het begin van de lijst te voegen. De tweede operatie levert het eerste element van een lijst en de derde operatie levert de lijst die bij het tweede element begint.

Vele typen kunnen worden beschouwd als beperkte versies van sequences. Een *stack* van X (of stapel van X) is een sequence van X met beperkte toegangsmogelijkheden. De toegestane operaties komen overeen met de LISP-operaties *cons*, *car* en *cdr*, maar hebben de namen

```
push          top          pop
```

Een *queue* verschilt alleen daarin van een stack, dat de operatie pop een element van het andere einde van de sequence neemt. De operaties op een queue heten meestal *insert* en *delete*. Een stack wordt soms een *lifo* (last-in first-out) genoemd, een queue *fifo* (first-in first-out).

Een andere beperkte versie van sequences die in de praktijk voorkomt is te vinden in invoer- en uitvoerfiles. Een invoer-file kan worden beschouwd als een queue zonder delete-operatie. Pascal definieert files formeel als sequences met beperkte operaties. Een van de sterkste punten van het besturingssysteem UNIX* is het feit dat daarin files worden behandeld als een gewone sequence van tekens met goed gedefinieerde, maar beperkte operaties. Het teken voor overgang naar een nieuwe regel is daarbij ook werkelijk een teken. Daardoor is een file in UNIX gewoon een sequence van tekens, niet een sequence van records. Door die aanpak worden de meeste invoer/uitvoer-operaties vereenvoudigd, omdat er alleen tekens worden gelezen en geschreven, geen records.

---

* UNIX is een handelsmerk van AT&T Bell Laboratories

Opgaven

1. Beschouw de programmeertalen die arrays van nul elementen toestaan. Hoeveel verschillende arraytypen hebben daarin nul elementen en hoeveel verschillende waarden bestaan er voor elk van die typen?

2. De operaties `+/` en `*/` leveren de som en het produkt van de elementen van een vector. Welke waarden moeten deze operaties opleveren als ze worden toegepast op een lege array? Hoe kan deze klasse van operaties zo worden gegeneraliseerd, dat ze ook op arrays van nul elementen toepasbaar zijn?

3. Beschouw de problemen die ontstaan bij de behandeling van willekeurige sequences van tekens, waarbij speciale tekenwaarden worden gebruikt om zaken als einde van een regel, einde van een record en einde van een file aan te geven. Ga ervan uit dat alle tekens al een eigen betekenis hebben. Bedenk een gegevenstype STRINGPLUS dat zulke speciale waarden toestaat zonder dat er tekens voor worden opgeofferd. Gebruik het type variant record of het type union uit Pascal, Ada of ALGOL 68 en geef een definitie van concatenatie en vergelijking voor het type STRINGPLUS.

4. Een complex getal kan worden gerepresenteerd als een array of als een record.

```
type COMPLEX_EEN is array (1..2) of REAL;

type COMPLEX_TWEE is
        record
                RE, IM: REAL;
        end record;
```

   Vergelijk deze beide representaties en geef de verschillen aan. Wat zijn de voor- en nadelen?

4

Andere typen

In dit hoofdstuk zien we typen die geen basistype zijn, maar ook geen samengesteld type. Veel programmeurs zijn misschien niet vertrouwd met deze typen, maar ze spelen toch een belangrijke rol en hebben invloed op vele kwesties rond gegevenstypen.

4.1 Pointers

In de traditie van gegevenstypen vormen variabelen, in tegenstelling tot waarden, geen deel van het typensysteem. Variabelen maken deel uit van het geheugenbeheer en worden niet als waarden beschouwd. Dit onderscheid is vooral lastig als ook pointers deel uitmaken van het typensysteem, want de waarde van een pointer is een variabele. ALGOL 68 vormt een uitzondering op deze traditie. In ALGOL 68 slaat de term *referentie* op een waarde die verwijst naar een geheugenplaats die weer een andere waarde bevat. Een gegevenstype zonder referentie is in ALGOL 68 een constante (omdat er geen geheugenplaats bij betrokken is). Een identifier met als type int kan in ALGOL 68 niet van waarde veranderen en kan dus ook niet aan de linkerkant voorkomen in een toekenningsopdracht. Een gegevenstype met referentie is in ALGOL 68 een variabele. Een integer variabele heeft dan als type `ref int`. Een type met twee of meer referenties is in ALGOL 68 een pointer-waarde. Een integer pointer heeft als type `ref ref int`. Beschouw deze declaraties in ALGOL 68:

```
int          CONSTANTE1 = 5;
ref int      VARIABELE1 = loc int := READINT;
ref ref int  POINTER1   = loc ref int;
int          CONSTANTE2 = VARIABELE1*CONSTANTE1;
```

De identifier CONSTANTE1 is de naam van een constante. Er kan geen nieuwe waarde aan worden toegekend. In tegenstelling tot Pascal, waarin de waarde van een constante tijdens het compileren bekend moet zijn, mogen constanten in ALGOL 68 een waarde hebben die pas tijdens het uitvoeren van het programma wordt berekend. Op de vierde regel van bovenstaand voorbeeld krijgt de constante CONSTANTE2 een waarde die afhangt van VARIABELE1, waarvan de startwaarde uit een externe file is gelezen.

Doordat in ALGOL 68 het begrip referentie is geïntroduceerd, zijn de concepten constante, variabele en pointer eenvoudiger geworden. Als er geen duidelijk beeld van waarden, variabelen en pointers is, kunnen er verwarrende situaties ontstaan. In de definitie van andere talen moet er iets aan die extra complexiteit worden gedaan. Neem een geïndiceerde variabele van een integer array. In de ene context kan deze een waarde voorstellen, maar in een andere context, bijvoorbeeld aan de linkerkant van een toekenningsopdracht of als argument dat by reference wordt doorgegeven, stelt de geïndiceerde array-variabele een zogenaamde integer *l-waarde* voor. In een taal waarin het begrip referentie geen deel uitmaakt van het typensysteem, is elke geïndiceerde integer array van het type integer. Er wordt dan geen verschil gemaakt tussen een geïndiceerd array-element dat als waarde gebruikt gaat worden en één dat als variabele gebruikt gaat worden. In de typenstructuur van ALGOL 68 is het duidelijk dat het indiceren van een referentie naar een integer array niet een integer oplevert, maar een referentie naar een integer. In die conventie zou het duidelijk toegestaan zijn zo'n expressie aan de linkerkant van een toekenningsopdracht te plaatsen. Op dezelfde manier levert de selectie van een veld van een referentie naar een record niet de waarde van dat veld, maar een referentie naar dat veld. In sommige talen is zowel call-by-value als call-by-reference toegestaan. In ALGOL 68 worden parameters formeel gesproken altijd by value doorgegeven, maar het doorgeven van een referentie is equivalent met call-by-reference. Het parametermechanisme van ALGOL 68 is dus eenvoudig, maar zeer flexibel.

Pointers werden in het begin van de jaren 60 in PL/I gebruikt om het behandelen van lijsten in een hogere programmeertaal mogelijk te maken. In die tijd had men nog geen goed begrip van typecontrole; in PL/I werd het type van pointers dan ook niet gecontroleerd, noch tijdens het compileren, noch tijdens het uitvoeren van het programma. In PL/I kan een pointer naar alles wijzen en heeft dus geen type. ALGOL 68 kwam met het idee pointers een type te geven, om zo te voorzien in dit gebrek aan typecontrole. Een pointer met een type kan alleen wijzen naar variabelen van één type. Een integer pointer kan alleen naar integer variabelen wijzen.

Met pointers ontstond er een nieuw probleem. Een *hangende pointer* is een pointer die wijst naar wat eens ten behoeve van een variabele in gebruik was, maar nu niet meer. Zo'n uitzonderingstoestand kan ontstaan als de variabele expliciet wordt vrijgegeven, of als een globale pointer bij het verlaten van een blok wijst naar een locale variabele. In ALGOL 68 kunnen hangende pointers niet tijdens het compileren worden ontdekt. In plaats daarvan moet er op de juiste plaatsen een controle worden ingebouwd die tijdens uitvoering van het programma toekenningen voorkomt die tot hangende pointers kunnen leiden. De toekenning van VARIABELE2 aan POINTER2 in het volgende programma levert een voorbeeld van een hangende pointer. Het sleutelwoord `pointer` betekent in dit voorbeeld 'wijst naar'; in ALGOL 68 wordt dit met `ref` aangegeven, in Ada met `access` en in Pascal met een pijltje naar boven (↑).

```
POINTER2: pointer INTEGER;
P: procedure is
        VARIABELE2: INTEGER;
begin
        VARIABELE2 := 5;
        POINTER2 := ADRES(VARIABELE2);
end P;
MAIN: procedure is
begin
        P;
        PRINT(POINTER2);
end MAIN;
```

In bovenstaand voorbeeld wordt aan VARIABELE2 de waarde 5 en aan POINTER2 het adres van VARIABELE2 toegekend. Maar na het verlaten van de procedure P wordt het geheugen voor de variabele VARIABELE2 vrijgegeven en blijft de pointer POINTER2 hangen. De gevolgen van later gebruik van deze hangende pointer zijn onvoorspelbaar. Zo kan het vrijgekomen geheugen gealloceerd zijn voor een andere variabele. In Pascal is er een nieuwe oplossing geïntroduceerd voor het probleem van de hangende referentie. Opdat tijdens het compileren kan worden vastgesteld dat er geen hangende referenties kunnen voorkomen, mogen pointers in Pascal alleen wijzen naar speciaal gecreëerde anonieme variabelen op de heap. In Pascal is de functie ADRES niet beschikbaar, zodat pointers niet naar andere variabelen kunnen wijzen. In dezelfde geest beschikt Ada ook niet over de functie ADRES.

Net als bij andere typen zou men ook bij het type pointer kunnen vragen wat de constanten, de waarden en de operatoren zijn. De waarde van een variabele van een pointertype is in feite het adres van een geheugenplaats waar een waarde van een of ander type is opgeslagen. De enige uitzondering is de nul-pointer,

die in bijna alle talen gebruikt wordt als pointerwaarde die niet naar een geheugenplaats wijst. Deze kan worden gebruikt voor niet geïnitialiseerde pointervariabelen en voor lege lijsten en bomen. De nulwaarde is meestal de enige letterlijke constante van het type pointer.

De belangrijkste functie op pointers is de allocatiefunctie, die een verse geheugenplaats zoekt. Een andere nuttige functie is de eerder genoemde functie ADRES. De meeste programmeertalen staan het vergelijken van pointers toe. Twee pointers worden als gelijk beschouwd als ze naar hetzelfde object wijzen. De meeste talen hebben geen andere operaties op pointers. De taal C is een uitzondering; daarin is het rekenen met pointers toegestaan. Arrays en pointers zijn in C conceptueel gelijk. Indiceren wordt in C hetzelfde opgevat als het optellen van een getal bij de betreffende pointer. Op die manier betekent `a[j]` de waarde in de geheugenplaats `a+j`.

4.2 Unions en variante records

Een union-type is geen samengesteld type, omdat het geen collectie van gegevens is. Het is wel een constructortype, omdat uniontypen opgebouwd worden met behulp van andere typen. Unions dienen een verscheidenheid aan doeleinden, zodat de één een iets andere opvatting van unions zal hebben dan de ander. Het komt erop neer dat een uniontype de waarden van een aantal andere typen, de *alternatief-typen*, combineert. Soms wordt het woord *variant* gebruikt in plaats van alternatief. Een uniontype met de twee alternatieftypen integer en teken heeft als waarden zowel de integers als de tekens. Als `x` een variabele is van dat type, dan kunnen aan `x` zowel integers als tekens worden toegekend. Maar `x` heeft op elk moment maar één waarde, die ofwel een integer, ofwel een teken is. In talen waarin tijdens de compilatie de typen worden gecontroleerd, zullen (in het algemeen) operatoren op integers en op tekens niet op `x` mogen worden toegepast, omdat `x` tijdens de uitvoering van het programma het verkeerde type zou kunnen hebben.

Unions worden soms gebruikt om geheugen te sparen of om aan hetzelfde object verschillende namen te geven (*alias*). Unions maken het mogelijk dat een aantal conceptueel verschillende variabelen de plaats van één variabele innemen. FORTRAN gebruikt equivalentie voor dat doel. COBOL beschikt voor hetzelfde doel over REDEFINES.

Een belangrijker doel van unions is het representeren van typen die kunnen bestaan uit verschillende categorieën van waarden die niet gemakkelijk als één enkel type kunnen worden gerepresenteerd. Neem een type, dat we SCORE zullen noemen, en waarvan de waarde een integer kan zijn of één van de drie niet-integer waarden ONBEKEND, VREEMD en ANDERS. Wie probeert een gewoon integer type te gebruiken voor het type SCORE, moet op een of andere manier de drie niet-integer waarden representeren. In het volgende voorbeeld wordt zo'n representatie gebruikt en worden de twee functies TEL1OP en TREK1AF gedefinieerd.

```
type SCORE is INTEGER;
constant VREEMD   = -100;
constant ONBEKEND = -101;
constant ANDERS   = -102;

TEL1OP: function (X: SCORE) return SCORE is
begin
        if X=VREEMD or X=ONBEKEND or X=ANDERS then
                return X;
        else
                return X+1;
        end if;
end TEL1OP;

TREK1AF: function (X: SCORE) return SCORE is
begin
        return X-1;
end TREK1AF;
```

Deze oplossing bevat een aantal duidelijke onjuistheden.

1. Het eerste probleem is dat bovenstaande oplossing aanneemt dat de getallen –100, –101 en –102 nooit als geldige scores kunnen voorkomen. Deze waarden worden hier dus voor een speciale betekenis gereserveerd.

2. Het tweede probleem is dat de functie TREK1AF de speciale waarden niet apart behandelt. De aanroep TREK1AF(ANDERS) levert bijvoorbeeld de waarde –103 af. Waar het om gaat is, dat de compiler geen typefouten kan ontdekken.

Een andere manier is een collectie van variabelen te gebruiken voor het opslaan van de informatie van een waarde van het type SCORE. Eén waarde zou uitsluitend voor geldige integer scores kunnen worden gebruikt, een tweede variabele voor de niet-integer variabelen en een derde om aan te geven of het om een

integer of een niet-integer score gaat. Deze oplossing verdeelt wat conceptueel één variabele is over drie variabelen, maar we kunnen die als volgt in één enkel record plaatsen:

```
type SCORE is
        record
             AANTAL:    INTEGER;
             PROBLEEM: (ONBEKEND, VREEMD, ANDERS);
             WELK:      (GOED, SLECHT);
        end record;

TEL1OP: function (X: SCORE) return SCORE is

begin
        if X.WELK = GOED then
                X.AANTAL := X.AANTAL+1;
        end if;
        return X;
end TEL1OP;

TREK1AF: function (X: SCORE) return SCORE is
begin
        X.AANTAL := X.AANTAL-1;
        return X;
end TREK1AF;
```

Dit voorbeeld is een verbetering ten opzichte van het vorige, omdat er geen speciale betekenis meer wordt gegeven aan sommige integer waarden. Merk op dat de functie TREK1AF nu correct werkt, ook al is die niet juist geschreven. De functie werkt correct omdat de velden PROBLEEM en WELK niet worden veranderd. Merk echter ook op dat het veld AANTAL niet op de juiste wijze wordt gebruikt als de score SLECHT is. Met een uniontype zouden deze typefouten ontdekt worden, maar bij een gewoon recordtype kan de compiler zulke fouten niet ontdekken.

Het blindelings gebruiken van een waarde van een uniontype alsof het een waarde van zo maar een type is, heeft typefouten en onjuist gebruik van operatoren tot gevolg. De vereiste typecontrole kan niet altijd tijdens het compileren worden gedaan. In de meeste situaties moet er ook enige run-time controle worden uitgevoerd. Daarvoor moet dan tijdens de uitvoering van het programma bepaalde informatie worden bijgehouden die het type aangeeft van de huidige waarde die in een variabele van een uniontype wordt bewaard. Die informatie heet een *discriminant*. In het vorige voorbeeld speelde WELK de rol van discriminant. Een toekenning aan een union-variabele moet zowel de waarde als de discriminant aanpassen. Bij elk gebruik van de waarde moet de discri-

minant worden geraadpleegd. Eén van de manieren daarvoor wordt geleverd door de discriminated case-opdracht.

Een *discriminated case-opdracht* is een besturingsopdracht met meerdere takken (een gegeneraliseerde conditionele opdracht), met één tak per alternatieftype. Tijdens het uitvoeren van het programma wordt de juiste tak bepaald aan de hand van de actuele waarde van de discriminant. We zullen verderop voorbeelden hiervan geven. Sommige programmeertalen hebben een andere methode; daarin worden *projectie*-operatoren gebruikt, die de waarde van een uniontype converteren naar één van de alternatieftypen, en *injectie*-operatoren, die een alternatieftype converteren naar een uniontype.

Een tweede belangrijke kwestie met betrekking tot unions is de precieze aard van een union. Bestaan er unions van twee integer typen? In ALGOL 68 kan een union geen onderling verwante typen bevatten; zo'n union wordt dan *incestueus* genoemd. Twee typen zijn verwant als het ene door middel van coërcie in het andere kan worden overgevoerd. Zo zijn in ALGOL 68 de typen `int` en `ref int` verwant en kunnen niet van dezelfde union deel uitmaken. Omdat elk type door coërcie in zichzelf kan worden overgevoerd, moet elk alternatieftype van een union verschillend zijn. Dit staat in tegenstelling tot de *discriminated union,* waarin geen beperkingen gelden voor het combineren van typen. Als twee alternatieftypen van een discriminated union gelijk zijn, kunnen ze toch van elkaar worden onderscheiden door middel van de discriminant. Een discriminated union lijkt dus meer op een disjuncte union dan op een gewone, verzamelingachtige union van het huis-tuin-en-keukensoort. Pascal en Ada hebben allebei de discriminated union in hun typensysteem opgenomen, en wel onder de naam *variant record.*

Variante records zijn discriminated unions zonder protectie van de discriminant. Een voorbeeld van een variant record is

```
type SCORE is
    record
        case WELK: (GOED, SLECHT) is
        when GOED   => AANTAL: INTEGER;
        when SLECHT => PROBLEEM:(ONBEKEND,VREEMD,ANDERS);
        end case;
    end record;

TEL1OP: function (X: SCORE) return SCORE is
begin
        if X.WELK = GOED then
                X.AANTAL := X.AANTAL+1;
```

```
        end if;
        return X;
    end TEL1OP;

    TREK1AF: function (X: SCORE) return SCORE is
    begin
        X.AANTAL := X.AANTAL-1;
        return X;
    end TREK1AF;
```

Het type SCORE is een record met ofwel een integer, ofwel de aanduiding van een probleemgeval. Het record bestaat uit drie componenten. De eerste component is de discriminant, die, zoals in het voorbeeld, met de selector WELK kan worden geselecteerd. De discriminant is van een opgesomd type en kan één van de beide waarden GOED en SLECHT hebben. De tweede component van het record kan worden geselecteerd met de selector AANTAL, maar alleen als de waarde van de discriminant GOED is. Op dezelfde manier kan de derde component, met selector PROBLEEM, alleen worden geselecteerd als de waarde van de discriminant SLECHT is. Omdat AANTAL en PROBLEEM niet tegelijkertijd in gebruik kunnen zijn, kunnen ze in hetzelfde stuk geheugen worden opgeslagen. Dit delen van dezelfde geheugenruimte door de alternatieven van een union is een belangrijke eigenschap in alle talen.

De functie TREK1AF functioneert niet goed als die ooit wordt aangeroepen met een score die SLECHT is. In sommige talen en implementaties wordt die fout niet gedetecteerd. Een compiler kan gemakkelijk speciale code toevoegen om vóór het gebruik van AANTAL te controleren of de discriminant de juiste waarde heeft. De discriminated case-opdracht wordt vaak gebruikt om die controle te combineren met een meervoudige vertakking. Herschreven met zo'n opdracht ziet de functie TREK1AF er als volgt uit:

```
    TREK1AF: function (X: SCORE) return SCORE is
    begin
        case X.WELK is
        when GOED =>
            X.AANTAL := X.AANTAL-1;
        when SLECHT =>
        end case;
        return X;
    end TREK1AF;
```

Bovenstaand voorbeeld laat het gebruik van de discriminated case-opdracht zien. Afhankelijk van het type van X wordt er één van de twee takken ingeslagen. Alleen als de waarde van het veld WELK gelijk is aan GOED wordt de eerste tak gekozen.

De union van ALGOL 68 is geen discriminated union, maar heeft wel een discriminant, die echter niet zichtbaar is voor de programmeur. Het beheer van de discriminant wordt geheel door het systeem verzorgd en volledige typecontrole is mogelijk. Beschouw het volgende voorbeeld:

```
type PROBLEEMTYPE is (ONBEKEND, VREEMD, ANDERS);
type SCORE is union (INTEGER, PROBLEEMTYPE);

TEL1OP: function (X: SCORE) return SCORE is
begin
        case X is
        when INTEGER =>
            X := X+1;
        end case;
        return X;
end TEL1OP;

TREK1AF: function (X: SCORE) return SCORE is
begin
        case X is
        when INTEGER =>
            X := X-1;
        end case;
        return X;
end TREK1AF;
```

Het belangrijkste element is hier de veiligheid betreffende de typen, die geboden wordt door de verborgen discriminant. Een toekenning aan een union-variabele gaat gepaard met een verborgen toekenning aan de discriminant. De case-opdracht gebruikt de verborgen discriminant om te bepalen welke tak er moet worden gekozen. Elke tak geeft de gelegenheid de waarde van één van de alternatieftypen als constante te gebruiken.

Variante records hebben twee belangrijke voordelen boven de unions van ALGOL 68. Ten eerste gaan variante records mooi samen met gewone records, die een fraaie syntactische weergave van ingewikkelde typen geven. Op de tweede plaats is een variant record een discriminated union, zodat elke combinatie van typen voor de varianten kan worden gebruikt, zelfs twee maal hetzelfde type. Neem het volgende voorbeeld dat zich lastig in ALGOL 68 laat vertalen.

```
type FIGUUR is
    record
         X_PLAATS: REAL;
         Y_PLAATS: REAL;
         case VORM: (VIERKANT, RECHTHOEK, CIRKEL, ELLIPS) is
         when VIERKANT =>  ZIJDE:               REAL;
```

```
        when RECHTHOEK => LENGTE, BREEDTE: REAL;
        when CIRKEL =>    DOORSNEE:        REAL;
        when ELLIPS =>    X_AS, Y_AS:      REAL;
      end case;
    end record;
```

Ada geeft een zodanige uitbreiding aan de variante records van Pascal, dat de veiligheid met betrekking tot de typen zeker is gesteld. In Ada kan de discriminant van een record met *variante delen* niet expliciet worden veranderd. Die wordt alleen veranderd als de variant wordt veranderd. Tevens wordt elke toegang tot een variant deel (zo nodig) tijdens het uitvoeren van het programma gecontroleerd. Dit type verschilt nog van de union uit ALGOL 68, doordat het in Ada mogelijk is dat er een run-time foutmelding DISCRIMINANT_ERROR optreedt. Zo'n foutmelding kan in ALGOL 68 niet voorkomen.

In bepaalde programma's kan het voorkomen dat een union-variabele altijd een waarde heeft van één bepaald alternatieftype. Dat gebeurt als er een variabele van een recursief gegevenstype is die nooit wordt veranderd. In de unions die we hierboven hebben gezien, hebben we niet aangenomen dat een variabele alleen waarden van één bepaald alternatieftype zou krijgen. Als we daar wel van uitgaan, dan kan het systeem precies de hoeveelheid geheugen alloceren die voor dat alternatief nodig is, in plaats van de hoeveelheid geheugen die nodig is voor het grootste alternatief. Die aanpak kan gemakkelijk worden uitgedrukt in de taal PL/I, die niet beschikt over unions of pointers met een type. Laten we eerst eens een gegevensstructuur in PL/I bekijken waarin niet op het zuinig gebruik van geheugen wordt gelet :

```
1 FIGUUR,
    2 X_PLAATS FLOAT BIN,
    2 Y_PLAATS FLOAT BIN,
    2 VORM FIXED BIN,
    2 VIERKANT,
        3 ZIJDE FLOAT BIN,
    2 RECHTHOEK,
        3 LENGTE FLOAT BIN,
        3 BREEDTE FLOAT BIN,
    2 CIRKEL,
        3 DOORSNEE FLOAT BIN,
    2 ELLIPS,
        3 X_AS FLOAT BIN,
        3 Y_AS FLOAT BIN;
```

In deze implementatie wordt voor elk alternatief apart ruimte gereserveerd in plaats van dat er ruimte door de alternatieven wordt gedeeld. Om deze verspilling van ruimte tegen te gaan, kunnen we de gegevensstructuur beschrijven als:

```
1 VIERKANT_FIGUUR,
    2 X_PLAATS FLOAT BIN,
    2 Y_PLAATS FLOAT BIN,
    2 VORM FIXED BIN,
        3 ZIJDE FLOAT BIN;

1 RECHTHOEK_FIGUUR,
    2 X_PLAATS FLOAT BIN,
    2 Y_PLAATS FLOAT BIN,
    2 VORM FIXED BIN,
        3 LENGTE FLOAT BIN,
        3 BREEDTE FLOAT BIN;

1 CIRKEL_FIGUUR,
    2 X_PLAATS FLOAT BIN,
    2 Y_PLAATS FLOAT BIN,
    2 VORM FIXED BIN,
        3 DOORSNEE FLOAT BIN;

1 ELLIPS_FIGUUR,
    2 X_PLAATS FLOAT BIN,
    2 Y_PLAATS FLOAT BIN,
    2 VORM FIXED BIN,
        3 X_AS FLOAT BIN,
        3 Y_AS FLOAT BIN;

1 FIGUUR,
    2 X_PLAATS FLOAT BIN,
    2 Y_PLAATS FLOAT BIN,
    2 VORM FIXED BIN;
```

In dit voorbeeld zou het vijfde record worden gebruikt als het gaat om een willekeurig record. Voor specifieke soorten van records zouden de eerste vier records worden gebruikt. Bij het gebruik van deze techniek moet de programmeur de records zorgvuldig zodanig inrichten dat de eerste velden identiek zijn. Deze methode werkt alleen omdat PL/I pointers zonder type heeft, die kunnen worden gebruikt om naar elk van de bovenstaande vijf records te wijzen. Als van één van de eerste vier records gebruik wordt gemaakt, kan de precieze hoeveelheid geheugen worden gealloceerd. Als de VORM van een record zal worden veranderd, dan moet de grootste van de records worden gealloceerd, zodat elke vorm in de gealloceerde geheugenruimte past.

In Ada kan deze aanpak ook worden gevolgd, maar met typeveiligheid. Het is in Ada mogelijk variabelen met of zonder discriminant te declareren. Als in Ada een record wordt gedeclareerd met een bepaalde discriminant, dan alloceert het systeem alleen de hoeveelheid geheugen die bij die discriminant nodig is. Dit wordt in Ada een *beperkt* (constrained) type genoemd, omdat een waarde

voor zo'n record alleen een waarde mag bevatten voor die ene waarde van de discriminant. Beschouw de volgende typedeclaratie in Ada:

```
type VORM is (VIERKANT, RECHTHOEK, CIRKEL, ELLIPS);

type FIGUUR (TYPE:VORM := VIERKANT) is
        record
            X_PLAATS: REAL;
            Y_PLAATS: REAL;
            case VORM is
            when VIERKANT  => ZIJDE:            REAL;
            when RECHTHOEK => LENGTE, BREEDTE: REAL;
            when CIRKEL    => DOORSNEE:         REAL;
            when ELLIPS    => X_AS, Y_AS:       REAL;
              end case;
        end record;

VIERKANT1, VIERKANT2: FIGUUR(VIERKANT);
OBJ1, OBJ2: FIGUUR;
```

Dit gegevenstype in Ada heeft het voordeel dat precies de benodigde hoeveelheid geheugen wordt gealloceerd en dat er een typeveiligheid geboden wordt die garandeert dat de waarden correct worden gebruikt. De declaraties voor VIERKANT1 en VIERKANT2 alloceren voldoende ruimte voor het opslaan van een vierkant. Aan deze variabelen kunnen alleen objecten met een vierkante vorm worden toegekend. De declaraties voor OBJ1 en OBJ2 alloceren genoeg ruimte voor het opslaan van elke waarde van het type FIGUUR.

Algemene typen

Enkele programmeertalen kennen de mogelijkheid van zogenaamde *algemene* (generic) procedures. Een algemene procedure is algemener bruikbaar omdat die werkt met parameters van een breder scala van gegevenstypen. We willen misschien een kwadraatfunctie zowel voor integers als voor reals. Daartoe kunnen we een tweetal functies INTKWADRAAT en REALKWADRAAT schrijven. Maar we zouden graag één procedure kunnen schrijven die zowel voor integer als voor real parameters werkt. We kunnen dat proberen met:

```
KWADRAAT: function (X:REAL) return REAL;
```

Als onze taal geen automatische conversie van integer naar real heeft, dan kunnen integers met deze functie niet worden gekwadrateerd. Als de taal wel automatische conversie van integer naar real heeft, dan werkt deze aanpak toch nog

niet goed, omdat het resultaat altijd real is. Het type union van ALGOL 68 werkt hiervoor wel; we kunnen iets schrijven als:

```
KWADRAAT: function (X: union (REAL, INTEGER))
                return union (REAL, INTEGER) is
begin
        case X
        when REAL    => ...
        when INTEGER => ...
        end case;
end KWADRAAT;
```

Een bezwaar van deze oplossing is dat het resultaat een union is in plaats van ofwel een real ofwel een integer. In andere talen zijn andere oplossingen mogelijk. PL/I geeft de mogelijkheid de procedure op verschillende punten binnen te komen, afhankelijk van het type van de parameters. EL1 beschikt over een constructie *oneof* die het mogelijk maakt dat een parameter van één van een aantal verschillende typen is, maar in tegenstelling tot ALGOL 68 en PL/I kunnen waarden van verschillende typen als resultaat worden afgeleverd. De meeste van deze methoden kunnen worden vervangen door een algemenere aanpak met meervoudig gedefinieerde operatoren: *overloading*. In Ada is overloading van functienamen toegestaan, zodat de programmeur twee verschillende functies KWADRAAT kan schrijven:

```
KWADRAAT: function (X:REAL) return REAL is
begin ... end

KWADRAAT: function (X:INTEGER) return INTEGER is
begin ... end
```

4.3 Recursieve typen

Een recursieve definitie van een gegevenstype is een definitie die naar zichzelf verwijst. Recursie kan een zeer natuurlijke manier zijn om bepaalde dingen uit te drukken. De recursieve definitie van een binaire boom is begrijpelijker dan een niet-recursieve definitie. Een binaire boom van *X* is ofwel leeg, ofwel een record bestaande uit drie dingen: een *X* en twee binaire bomen met de namen links en rechts. Andere op natuurlijke wijze optredende recursieve typen zijn o.a. lijsten, sequences en allerlei grafen. De *s*-expressie uit LISP was een van de eerste algemeen gebruikte recursieve typen die in programmeertalen voorkwamen. Een *s*-expressie is ofwel een atoom, ofwel een lijst *s*-expressies tussen haakjes, van elkaar gescheiden door spaties. Hoewel de gehele getallen recur-

sief kunnen worden gedefinieerd, vormen ze in de meeste programmeertalen een basistype. Een natuurlijk getal (een geheel getal groter dan 0) is ofwel 1, ofwel de opvolger van een natuurlijk getal. De opvolger van een getal wordt verkregen door er 1 bij te tellen. Op dezelfde manier kunnen lijsten, strings, sequences, stacks en queues gemakkelijk als recursieve typen worden gedefinieerd. Maar sommige talen leveren al speciale voorzieningen voor deze typen.

Sommige typen kunnen zonder recursie niet worden uitgedrukt. Het is bijvoorbeeld niet mogelijk een geketende lijst of een binaire boom te definiëren met alleen de constructortypen record, array, function, union en pointer. Alle binaire bomen van integers met diepte 3 kunnen als volgt worden uitgedrukt:

```
type BOOM0 is VOID;                -- lege boom
type BOOM1 is
    record
        LINKS, RECHTS: BOOM0;  -- eindknoop
        INFO: INTEGER;
    end record;

type BOOM2 is
    record
        LINKS, RECHTS: union (BOOM0, BOOM1);
        INFO: INTEGER;
    end record;

type BOOM3 is
    record
        LINKS, RECHTS: union (BOOM0, BOOM1, BOOM2);
        INFO: INTEGER;
    end record;
```

Dit is een buitengewoon vervelende manier om met bomen te werken. Lijsten en bomen kunnen zonder recursie niet worden uitgedrukt, omdat het willekeurig grote gegevensstructuren zijn die niet met eindige typeconstructors kunnen worden gerepresenteerd. De grotere uitdrukkingskracht van recursieve typen maakt een aanzienlijk ingewikkelder geheugenbeheer nodig dan voor niet-recursieve typen. Voor een variabele van het type BOOM3 in bovenstaand voorbeeld moet zoveel geheugenruimte worden gealloceerd dat elke boom van diepte 3 erin past. Bij gebruik van dezelfde techniek voor willekeurige bomen (recursief gedefinieerd) zou er voor een boomvariabele een oneindige hoeveelheid geheugen nodig zijn:

```
type BOOM is
        record
                LINKS, RECHTS: BOOM;
```

```
                INFO: INTEGER;
        end record;

WORTEL: BOOM;
```

Voor de variabele WORTEL zou volgens deze definitie geheugenruimte moeten worden gealloceerd die groot genoeg is voor twee bomen en een integer. Maar voor elke boom is er weer ruimte voor twee bomen en een integer nodig. Met behulp van het type pointer (of access) kunnen we dit geheugenprobleem omzeilen door een niveau van indirectie in te bouwen. Elke pointer gebruikt dezelfde hoeveelheid geheugen, onafhankelijk van de aard van het object waarheen de pointer wijst; daardoor verbruiken recursieve typen met pointers niet oneindig veel geheugen. De bovenstaande definitie van een boom kan dan herschreven worden tot

```
type BOOM is
        record
                LINKS, RECHTS: pointer BOOM;
                INFO: INTEGER;
        end record;

WORTEL: BOOM;
```

In dit voorbeeld heeft de variabele WORTEL evenveel geheugen nodig als twee pointers en een integer. Bij het opbouwen van een boom moet natuurlijk dynamisch geheugen worden gealloceerd naarmate er meer knopen worden toegevoegd. Een klein bezwaar van deze definitie van bomen is, dat de lege boom niet kan worden gerepresenteerd. Een kleine verandering van de definitie lost dat probleem op:

```
type BOOM is
    pointer
        record
            LINKS, RECHTS: BOOM;
            INFO: INTEGER;
        end record;

WORTEL: BOOM;
```

De variabele WORTEL vereist nu alleen nog geheugenruimte voor een enkele pointer; de lege boom wordt op een natuurlijke manier voorgesteld door de nulpointer.

Om ervoor te zorgen dat recursieve typen goed kunnen worden opgeborgen, leggen de meeste programmeertalen aan recursieve typen beperkingen op.

Pascal en Ada eisen dat elk recursief type van een pointer is voorzien, zodat elk recursief type een eindige geheugenruimte beslaat. ALGOL 68 is minder streng en verlangt ofwel een pointer, ofwel een indirectie via een procedure. De volgende recursieve gegevenstypen zijn in ALGOL 68 dan ook geoorloofd.

```
type C is function (F:C) return C;

type FUNC is function (A:INTEGER; F:FUNC)
                return INTEGER;
FACT: FUNC is
begin
        if A<2 then
                return 1
        else
                return A*F(A-1)
        end if;
end FACT;

begin
        FACT(5, FACT)
        ...
```

Het is interessant dat met het recursieve gegevenstype FUNC een recursieve faculteitsfunctie kan worden gedefinieerd die zelf geen expliciete recursie bevat.

4.4 Functies en procedures als typen

Een procedure of subroutine is een programma dat door een ander programma wordt aangeroepen. Een procedure heeft in het algemeen dezelfde opbouw als het hoofdprogramma. Een aanroep van een procedure kan ook parameters doorgeven. Meestal is er een onderscheid tussen routines die een waarde afleveren en andere routines. Sommigen gebruiken het woord *procedure* voor routines die geen waarde afleveren en *functie* voor routines die wel een waarde afleveren. Een aanroep van een procedure is dan een opdracht, een aanroep van een functie een expressie.

Functies en arrays hebben gelijksoortige eigenschappen. Ze worden op dezelfde manier gebruikt en hun gedrag is soms gelijk. Vele talen hebben voor het aanroepen van een functie zelfs dezelfde syntaxis als voor het indiceren van een array. Andere talen, zoals Pascal en ALGOL 68, gebruiken vierkante haken voor indexering van arrays en ronde haakjes voor het aanroepen van een functie. Arrays kunnen verscheidene indices van verschillende typen hebben voor

het selecteren van een bepaald element van de array. Functies kunnen verscheidene parameters van verschillende typen hebben om een bepaalde waarde van de functie te berekenen. In zeker opzicht lijkt het verschil alleen te zitten in de implementatie, niet in het gebruik van het type. Als abstract gegevenstype beschouwd zijn arrays en functies wellicht gelijk. In het gebruik is de keuze een kwestie van efficiëntie, waarbij de klassieke afweging van tijd tegen ruimte wordt gemaakt. In welke opzichten verschillen beide nog meer van elkaar? In tegenstelling tot een array kan een functie een parameter hebben waarvan het type niet eindig is. Het type van arrayindices is altijd eindig. Ook al in tegenstelling tot arrays kunnen functies zij-effecten hebben en van globale variabelen afhangen. Ook zonder zij-effecten en globale variabelen kan een functie lokale statische variabelen hebben en zich zo een *historie* van vorige aanroepen herinneren, waardoor de waarde die de functie aflevert, weer kan worden beïnvloed.

Is het redelijk om procedures en functies als gegevenstypen te beschouwen? Eén van de eerste talen die procedures en functies de eerste-klassestatus gaven was ALGOL 68. ALGOL 68 maakt geen onderscheid tussen procedures en functies. In ALGOL 68 leveren procedures de waarde `nil` af. Het is in ALGOL 68 mogelijk procedure-variabelen en arrays van procedures te gebruiken, men kan een procedure als parameter doorgeven en er zijn zelfs procedure-constanten. Er zijn twee operaties die op procedures kunnen worden toegepast: de toekenning en de aanroep. Het type van een procedure is in ALGOL 68 een typeconstructor die is gebaseerd op de typen van alle parameters en op het type van het resultaat. Als de procedure met bepaalde parameters wordt gebruikt, dan moet het type van de parameters in overeenstemming zijn met het type van de procedure. Ook als een procedure wordt toegekend of als parameter wordt doorgegeven, moeten de typen kloppen. Die eis zorgt ervoor dat elke procedure tijdens de uitvoering van het programma alleen parameters van het verwachte type krijgt te verwerken. In sommige talen (zoals de oorspronkelijke versie van Pascal) is het begrip procedure als type niet goed ontwikkeld. Daardoor kunnen er gaten ontstaan in de typecontrole. Als het type van een procedure alleen is gebaseerd op het type van het resultaat en geen rekening houdt met het type van de parameters, dan kan het voorkomen dat, door het toekennen en doorgeven van procedures, een procedure wordt toegepast op gegevens van een verkeerd type, zonder dat dat wordt gecontroleerd tijdens het compileren (en waarschijnlijk evenmin tijdens het uitvoeren van het programma).

ALGOL 68 kent procedure-constanten. Die worden ook wel *anonieme* procedures genoemd omdat er aan zo'n procedure geen naam is verbonden. De

LAMBDA-constructie in LISP is een voorbeeld van een anonieme procedure. Omdat Ada geen anonieme procedures heeft, zullen we ze aan de taal toevoegen; we zullen dan alle volgende voorbeelden volgens de syntaxis van Ada geven. Als er geen anonieme procedures zijn, moet elke procedure een naam krijgen. In het volgende voorbeeld gaan we uit van een procedure (hier VOOR_ELKE genoemd) die een andere procedure toepast op elk element van een lijst. Om elk element van de lijst af te drukken zouden we deze expressie kunnen laten uitvoeren:

```
VOOR_ELKE (A_LIJST, PRINT_ELEMENT)
```

We definiëren de afdrukprocedure dan zo:

```
PRINT_ELEMENT: procedure (ELEMENT: ELEMENTTYPE) is
begin
        PRINT_REGEL ( ELEMENT.NAAM, ELEMENT.WAARDE)
end PRINT_ELEMENT;
```

Als PRINT_ELEMENT nergens anders wordt gebruikt, dan is het zinvol de procedure PRINT_ELEMENT alleen betekenis te laten hebben in de opdracht waarin hij voorkomt. Het is dan ook niet meer nodig de procedure een naam te geven. Het voorbeeld ziet er dan zo uit:

```
VOOR_ELKE(LIJSTA,
    procedure (ELEMENT: ELEMENTTYPE) is
    begin PRINT_REGEL(ELEMENT.NAAM, ELEMENT.WAARDE) end
);
```

De expressie die met `procedure` begint en met `end` eindigt is een procedureconstante. Deze heeft geen naam, maar is een goed gedefinieerde waarde van het type procedure. De constante wordt als parameter aan een andere functie doorgegeven, maar zou ook in een toekenningsopdracht kunnen voorkomen of expliciet kunnen worden aangeroepen. Het concept anonieme procedure is voor velen misschien nieuw, maar het is in de lambdacalculus en in de wiskunde al vele jaren in gebruik. De kwestie van anonieme procedures is voor gegevenstypen alleen in zoverre van belang, dat de status van het type procedure erdoor wordt bepaald. Als er in een taal wel manieren zijn om constanten van de typen getal, string, array, record en dergelijke weer te geven, maar niet voor het type procedure, dan zijn procedures tweede-klassepassagiers.

De meeste talen kenmerken alleen het aanroepen en soms nog het toekennen als operaties op procedures. Een fundamentele operatie (althans in de wiskunde) is

compositie. De compositie van twee functies levert weer een functie. Stel dat F en G integer functies zijn met één integer parameter. De compositie van F en G is een derde functie die hetzelfde resultaat heeft als het aanroepen van G, gevolgd door het aanroepen van F met als parameter het resultaat van G (dus F(G(x))). De meeste programmeertalen bevatten geen functie-compositie en in de meeste gevallen kan compositie ook niet in de taal worden uitgedrukt. Men kan proberen de operatie als volgt uit te drukken:

```
type FUNC_UNAIR is function (INTEGER) return INTEGER;

COMPOSITIE: function (F,G: FUNC_UNAIR) return FUNC_UNAIR is
        SAMENGESTELD: function (X:INTEGER) return INTEGER
        begin return F(G(X));
        end SAMENGESTELD;

begin
        return SAMENGESTELD;
end COMPOSITIE;
```

Het probleem dat zich met deze functie in normale programmeertalen voordoet, is dat de parameters van de functie COMPOSITIE bewaard moeten blijven nadat de functie al is verlaten. In de normale situatie worden zowel de parameters als de locale variabelen opgeruimd, omdat ze na afloop niet meer nodig zijn. In bovenstaand voorbeeld moet de functie zich na afloop F en G nog herinneren, omdat het afgeleverde resultaat in de toekomst nog gebruik maakt van deze twee waarden. Dit probleem met procedures vormt de tegenhanger van het probleem van de hangende referentie (zie paragraaf 4.1). Het *retentiemodel* voor geheugenbeheer is een van de voorstellen geweest voor het oplossen van het probleem van de hangende referentie en van de hangende procedure. In het retentiemodel worden alle locale variabelen en parameters bewaard die in de toekomst nog nodig kunnen zijn, in plaats van dat ze bij het verlaten van de functie worden opgeruimd.

Nog een laatste opmerking over syntaxis is op zijn plaats. Zowel Ada als Pascal hebben een speciale syntaxis voor het declareren van functies en procedures. Zodra functies en procedures eerste-klassepassagiers worden, kunnen ze op dezelfde manier worden gedefinieerd als elke andere constante of variabele. De functie SAMENGESTELD zou dan bijvoorbeeld zo kunnen worden gedefinieerd:

```
SAMENGESTELD: UNAIR := begin return F(G(X)); end;
```

Merk ook op dat we de functie SAMENGESTELD eigenlijk geen naam behoeven te geven, zodat we de functie COMPOSITIE zo hadden kunnen definiëren:

```
COMPOSITIE: function (F,G: UNAIR) return UNAIR is
begin
        return  function (X: INTEGER) return INTEGER is
                begin return F(G(X)); end;
end COMPOSITIE;
```

Procedurele gegevenstypen

In de vorige paragrafen hebben we niet veel voorbeelden gegeven van het gebruik van de verschillende gegevenstypen voor het opbouwen van gegevensstructuren. Omdat het gegevenstype procedure maar zelden als gegevensstructuur wordt gebruikt, is het instructief een paar procedurele gegevensstructuren in detail te bekijken. Het gegevenstype procedure wordt wel vaak als *deel* van een gegevensstructuur gebruikt, zoals bij arrays van procedures, maar dat bedoelen we niet. We zullen juist een voorbeeld laten zien van hoe procedures gebruikt kunnen worden om gegevens te structureren.

Laten we eens kijken naar het gegevenstype 'verzameling van integers'. Als we ervan uitgaan dat onze programmeertaal geen tevoren gedefinieerd type verzameling heeft, moeten we nagaan hoe we verzamelingen van integers kunnen representeren. In vele toepassingen worden voor het representeren van verzamelingen *bit-maps* gebruikt. In een bit-map wordt voor elk mogelijk element van een verzameling één bit gealloceerd. Als de bit aanstaat komt het overeenkomstige element in de verzameling voor. Voor een verzameling van integers is een bit-map niet de geëigende representatie, omdat het aantal elementen dat tot de verzameling kan behoren, te groot is. Zelfs als we het bereik van de integers dat tot de verzameling kan behoren, zouden beperken tot het gebied van één tot een miljoen, dan zouden we nog een bit-map nodig hebben ter lengte van een miljoen bits. Een andere gebruikelijke representatie is een geketende lijst van de elementen die in de verzameling voorkomen. Voor het representeren van de verzameling {3,5,7} zou dan een geketende lijst van drie elementen worden opgebouwd. Het is dan niet van belang in welke volgorde de elementen voorkomen. Deze representatie is op zijn plaats als de verzamelingen niet te groot worden. Voor het representeren van de verzameling van alle even getallen tussen één en een miljoen zou zo zelfs meer ruimte nodig zijn dan in de representatie met een bit-map. (Zie paragraaf 9.1 voor meer informatie over representaties van verzamelingen.)

Een procedurele representatie voor verzamelingen heeft niet de nadelen van de twee vorige representaties. Een procedurele representatie van een verzameling

is een Booleaanse functie met één integer parameter. Als het getal dat als parameter wordt doorgegeven tot de verzameling behoort, levert deze Booleaanse functie true af, en anders false. De lege verzameling wordt gerepresenteerd door een functie die altijd false aflevert:

```
type VERZAMELING is function (X: INTEGER) return BOOLEAN;

LEEG: VERZAMELING := begin return FALSE; end;
```

Ook andere verzamelingen, zoals {4,7} en de verzameling van even getallen, kunnen gemakkelijk worden gerepresenteerd:

```
VIER_EN_ZEVEN: VERZAMELING := begin return (X=4) or (X=7); end;
EVEN: VERZAMELING := begin return (X mod 2) = 0; end;
```

De bovenstaande voorbeelden laten zien dat het eenvoudig genoeg lijkt om constante verzamelingen te representeren. Operaties op verzamelingen moeten ook kunnen worden gedefinieerd. Als het geheugen volgens het retentieschema wordt beheerd, kunnen operaties gemakkelijk worden gedefinieerd. We geven hieronder de definitie voor de lidmaatschaps-, de toevoegings- en de doorsnijdingsoperatie. Merk op dat de toevoegingsoperatie beschreven wordt in een functie met een naam en de doorsnijdingsoperatie met een anonieme functie.

```
LID: function (X:INTEGER; V: VERZAMELING) return BOOLEAN is
begin
        return V(X);
end LID;

VOEG_TOE: function (Y:INTEGER; V:VERZAMELING) return
        VERZAMELING is
        RESULTAAT: VERZAMELING:=
        begin return (X=Y) or else V(Y); end RESULTAAT;
begin
        return RESULTAAT;
end VOEG_TOE;

DOORSNEE: function (V1,V2: VERZAMELING) return VERZAMELING is
begin
        return  function (X:INTEGER) return BOOLEAN is
                begin return V1(X) and then V2(X); end;
end DOORSNEE;
```

Procedurele gegevenstypen hebben een aantal nadelen. Ze zijn in de meeste talen onhandig of moeilijk te implementeren. Ze kunnen inefficiënt zijn, vooral als er vele operaties worden gebruikt voor het weergeven van omvangrijke gegevens. In sommige gevallen kunnen bepaalde operaties niet worden geïmple-

menteerd. Als bijvoorbeeld het type verzameling door een procedureel type wordt gerepresenteerd, is een procedure VOOR_ELKE of een dergelijke functie niet mogelijk. Dat komt doordat het onmogelijk of althans zeer duur is om alle elementen van een verzameling op te sommen. Het is misschien wel goedkoop om van een bepaald element vast te stellen of het tot een verzameling behoort, maar alleen door middel van uitputtend zoeken kunnen alle elementen van een willekeurige verzameling worden gevonden.

4.5 Andere typen

Er zijn vele andere gegevenstypen bedacht die niet in één van de vorige categorieën vallen. In bepaalde vormen van gegevensverwerking kunnen deze typen een nuttige rol spelen. We zullen een paar van deze ongebruikelijke typen in deze paragraaf kort behandelen.

Patronen in SNOBOL

SNOBOL is een programmeertaal die speciaal is ontworpen voor het manipuleren van strings. Eén van de concepten, het zogenaamde *patroon*, ontwikkelde zich tot een gegevenstype van SNOBOL 4. In ICON, een nieuwere programmeertaal, wordt de SNOBOL-traditie voortgezet en worden enkele van de ideeen achter pattern-matching gegeneraliseerd tot wat in ICON *doelgerichte evaluatie* wordt genoemd.

Een patroon specificeert een klasse van strings. Bij vergelijking met een bepaalde string kan een patroon slagen of falen. Het patroon is een soort grammatica. Een grammatica en een patroon definiëren beide een formele taal, wat gewoon neerkomt op een verzameling strings. Zowel op grammatica's als op patronen is de lidmaatschapstest de belangrijkste operatie. In SNOBOL wordt het testen op lidmaatschap *pattern-matching* genoemd, bij grammatica's heet het ontleden of *parsen*. Een succesvolle pattern-match (of parsing) treedt op als de string tot de taal behoort, anders faalt de operatie.

De volgende lijst geeft enkele van de fundamentele mogelijkheden voor pattern-matching in SNOBOL:

'ABC'	alleen succes met 'ABC'
LEN(4)	succes met elke string van lengte 4

x | y	succes met string die gelijk is aan string x of aan string y
x y	succes met elke string waarvan het eerste deel slaagt met x en het tweede met y
'A'('B' | 'C')	succes met 'AB' en met 'AC'

De eerste twee regels geven voorbeelden van patroonconstanten. Ze geven beide een vaste verzameling strings weer. In het eerste voorbeeld bevat de verzameling slechts één string. In het tweede voorbeeld behoren alle strings van vier tekens tot de verzameling. De volgende twee regels geven voorbeelden van operatoren. De eerste operator, *alternering* genoemd, kan willekeurige patronen als operanden hebben. Ook de tweede operator, *concatenatie* genaamd, kan als operanden willekeurige patronen hebben. De laatste regel geeft een voorbeeld van een expressie waarin deze operatoren allebei voorkomen. Concatenatie heeft de hoogste prioriteit, zodat er in dit voorbeeld haakjes nodig zijn (anders zou 'AB' of 'C' tot succes leiden). Patronen mogen aan variabelen worden toegekend, en uit eenvoudige patronen kunnen complexere patronen worden gevormd. In het volgende voorbeeld worden er patronen toegekend aan KLINKER, WOORD en ZIN.

```
KLINKER = 'A' | 'E' | 'I' | 'O' | 'U'
WOORD = 'P' KLINKER ('L' | 'T') | 'B' KLINKER 'S'
               | ('M' | 'L') KLINKER 'ST'
ZIN = WOORD ' ' WOORD ' ' WOORD
```

Het patroon ZIN definieert zinnen als 'BAS LUST PIL' en 'BES MIST PIT'. Omdat dit dynamische toekenningen zijn, is de volgorde van de toekenningen van belang. Als de derde toekenning vóór de eerste zou staan, zou ZIN alleen succes opleveren met twee spaties (want WOORD wordt, net als andere variabelen in SNOBOL, met de lege string geïnitialiseerd).

Er zijn nog veel meer dingen mogelijk met SNOBOL-patronen. Het is bijvoorbeeld prettig als we de substrings die succes opleveren te weten kunnen komen; daarom kan in SNOBOL elke geheel of gedeeltelijk overeenkomende string aan een variabele worden toegekend. Ook recursieve patronen zijn mogelijk, maar daarmee is in SNOBOL moeilijker te werken. Door deze extra problemen zijn SNOBOL-patronen moeilijk te formaliseren.

Verzamelingen

Een verzameling is een fundamentele wiskundige abstractie. Slechts weinig programmeertalen hebben verzamelingen als gegevenstype. Zelfs Ada, een moderne taal die op Pascal is gebaseerd, heeft geen verzamelingen. Pascal, de enige algemeen gebruikte taal met verzamelingen, kent alleen verzamelingen van opgesomde typen (of deelbereiken). Dat is een heel goede beperking in Pascal, geheel in overeenstemming met de doelstelling van eenvoudige en efficiënte implementatie. De verzamelingen van Pascal kunnen eenvoudig en efficiënt worden geïmplementeerd als bit-maps.

Implementaties worden gecompliceerder als het begrip verzameling wordt gegeneraliseerd tot alle typen. Voor typen met een zeer groot aantal waarden kan de bit-map onnodig lang worden. Typen met een oneindig aantal elementen kunnen niet door middel van een bit-map worden gerepresenteerd. In bepaalde gevallen vormt een dynamische gegevensstructuur een betere representatie, maar voor zulke structuren is een dynamische vorm van geheugenbeheer nodig. Dit soort opslagkwesties zijn gedeeltelijk de verklaring waarom het type verzameling in de meeste programmeertalen niet is opgenomen.

De programmeertaal SETL is op verzamelingen gebaseerd. Als basistypen zijn de gebruikelijke typen aanwezig, maar de drie fundamentele typeconstructors zijn *tupel*, *verzameling* en *afbeelding*. Een tupel is een heterogene array van variabele lengte. Uit die definitie blijkt al dat SETL run-time typecontrole en dynamisch geheugenbeheer heeft. Het type afbeelding bestaat uit een verzameling paren (waarbij een paar een tupel van lengte 2 is). De eerste elementen van alle paren vormen samen het domein en de tweede elementen het bereik (de range).

Een afbeelding kan als functie worden gebruikt. Het veranderen van een bestaande afbeelding is toegestaan, zodat een afbeelding lijkt op een array met een vrij gekozen indextype. In SETL kan men over elk constructortype itereren; de syntaxis daarvoor lijkt op die uit de wiskunde.

Typen als type

De programmeertalen EL1 en Russell zijn slechts twee van de vele talen die een gegevenstype hebben dat we hier *type* zullen noemen. De waarden van dit type zijn alle typen van de programmeertaal. Zo'n type verschaft een flexibele methode voor het uitdrukken van procedures waarvan de parameters en ook het

resultaat van verschillende typen kunnen zijn. Het maakt ook dynamische gegevenstypen mogelijk, dat wil zeggen dat tijdens de uitvoering van het programma nieuwe typen kunnen worden geconstrueerd.

De constanten van een type type zijn de basistypen en de operatoren van een type type zijn de constructortypen. INTEGER en CHARACTER zijn dus twee constanten en ARRAY is een functie van het type type. Ada is niet bijzonder geschikt voor het type type; laten we daarom zelf een systeem van gegevenstypen met een type type verzinnen. Als basistypen nemen we INTEGER, CHARACTER en TYPE. Laten we als constructortypen kiezen: Cartesisch produkt, discriminated union, functie en verzameling, die we achtereenvolgens noemen: RECORD, UNION, FUNCTION en SET. Elke typeconstructor kan elk gewenst aantal argumenten ongelijk nul hebben, behalve de constructor SET, die maar één argument kan hebben. Type-expressies zijn nu gewoon expressies die uit deze constanten en constructors zijn opgebouwd. Hier is een voorbeeld van declaraties waarin van het type type wordt gebruik gemaakt:

```
X: INTEGER := 5;
Z: RECORD(INTEGER, INTEGER) := (4,5);
T: TYPE := RECORD(INTEGER, CHARACTER);
type FUNC is RECORD(TYPE, INTEGER);
ARRAY: function (INDEX, ELEMENT: TYPE) return TYPE;
STRING: TYPE := ARRAY(INTEGER, CHARACTER);
S: STRING;
```

Deze manier van programmeren introduceert plotseling vele nieuwe ideeën die tot nu toe niet in beschouwing zijn genomen. Merk op dat er in bovenstaand voorbeeld geen verschil zou behoeven te bestaan tussen gewone typedeclaraties (zoals van FUNC) en declaraties van het type type (zoals van T). Er kunnen ook nieuwe typeconstructors, zoals ARRAY en STRING, worden gedefinieerd. De precieze aard en mogelijkheden van het type type lopen van taal tot taal uiteen. Eén van de moeilijkheden met deze manier van programmeren is dat er geen duidelijk verschil meer is tussen acties die worden uitgevoerd tijdens het compileren, en run-time acties.

Opgaven

1. In de taal C zijn pointers en arrays in feite gelijk. Beschouw de declaraties

```
VAR1: array (1..10) of INTEGER;
VAR2: pointer INTEGER;
```

VAR1 en VAR2 zijn equivalent, met de volgende uitzonderingen:

a. VAR1 is een constante en kan geen nieuwe waarde krijgen. De array-elementen van VAR1 zijn variabelen, die wel een nieuwe waarde kunnen krijgen.

b. De declaratie van VAR2 alloceert geheugenruimte voor de pointer. Met een aparte opdracht wordt een serie van tien integer variabelen gealloceerd.

VAR1 en VAR2 worden beide als een geheugenadres beschouwd, en indices worden beschouwd als pointer-expressies die een adres berekenen. Wat voor invloed heeft deze opvatting van arrays op het programmeren? Zijn er belangrijke conceptuele verschillen tussen traditionele arrays en de arrays van C die invloed hebben op het beeld dat een programmeur heeft van typen, waarden en geheugen?

2. We hebben niet gesproken over operatoren voor het type procedure. Geef mogelijke interpretaties van een gelijkheidsoperator voor procedures. Voor welke van deze interpretaties kan een implementatie worden gegeven?

3. Waarom zijn verzamelingen in de meeste programmeertalen geen centraal gegevenstype?

4. Beschouw het type type en de volgende definitie van een type lijst.

```
LIJST: function (X: TYPE) return TYPE is
begin
        return RECORD(X, LIJST(X));
end LIJST;
```

Welke problemen kunnen zich in een programmeertaal voordoen met deze typeconstructor? Is het mogelijk deze functie te herschrijven om die problemen te voorkomen?

Literatuur

Wexelblat (1987) is een bundel artikelen over de geschiedenis van de eerste programmeertalen. In inleidende boeken over de principes van programmeer-

talen, zoals Nicholls (1975) en Tennent (1981), is meer materiaal te vinden. Tennent (1973) en Fleck (1978) behandelen het formaliseren van stringpatronen. Reynolds (1970) behandelt in de taal GEDANKEN het gebruik van functies voor het selecteren van gegevensstructuren. Procedurele gegevenstypen worden in meer detail behandeld in Reynolds (1975). Verdere informatie over typen als type en over dynamische gegevenstypen is te vinden in Goodwin (1980) en in Demers et al. (1978, 1980a). De oorsprong van het idee van eersteklassepassagiers wordt gevormd door het orthogonaliteitsconcept van ALGOL 68 (Van Wijngaarden, 1975). In een latere publicatie geven Demers en anderen (1980b) een krachtige verdediging van *type-compleetheid*. Bij het uitbreiden van APL worden door Gull en Jenkins (1979) vele kwesties rond de betekenis van het begrip type met betrekking tot arrays onder de loep genomen en beoordeeld.

Deel II

Kwesties rond gegevenstypen

5 Typecontrole

5.1 Typefouten

Een *typefout* treedt op als een operatie een waarde van een verkeerd type te verwerken krijgt. De optel-operator voor integers verwacht bijvoorbeeld integer waarden. Als een integer optel-operatie een real waarde als operand krijgt, treedt er een typefout op. Als er geen typecontrole zou zijn, zou de integer optel-operator blindelings de beide operanden bij elkaar tellen, alsof het twee integers waren. De real waarde zou abusievelijk geïnterpreteerd worden als integer waarde. Die verkeerde interpretatie maakt een groot verschil en is machine-afhankelijk. Op een VAX wordt de real waarde 1.0 misverstaan als de integer waarde 16512. Elke operator verwacht één of meer operanden van bepaalde typen en levert een waarde af van een bepaald type. Als de operanden van een verkeerd type zijn, is er sprake van een typefout.

Een andere bron van typefouten is het onjuiste gebruik van variabelen. Een *getypeerde variabele* is een variabele waarin waarden van één bepaald type kunnen worden opgeslagen. Het onjuiste gebruik van variabelen komt alleen voor in talen met getypeerde variabelen. Elke poging om een waarde van het verkeerde type op te slaan, behoort een typefout te geven. Als aan een integer variabele een waarde wordt toegekend waarvan het type niet integer is, treedt er een typefout op.

Bepaalde fouten die traditioneel niet als typefouten worden beschouwd, kunnen toch typefouten worden als we de verzameling van typen veranderen. De fout

'delen door nul' treedt op als het tweede operand van de delingsoperator nul is. Omdat het resultaat rekenkundig niet gedefinieerd is, treedt er een fout op. Als we het type getal verdelen in de getallen ongelijk nul aan de ene kant en nul aan de andere, dan kan het type van het tweede operand van de deling opgevat worden als het type 'ongelijk nul'. Volgens die opvatting is deling door nul een typefout. Andere fouten die tot deze categorie behoren zijn het verlagen van het referentieniveau van de nulpointer, het op een lege lijst toepassen van de operator die het eerste element of de rest van een lijst oplevert en het verwijderen van het eerste element van de lege stack.

Fouten betreffende een bereik of deelbereik worden soms ook als typefouten beschouwd. Zoals eerder beschreven wordt een integer deelbereik door middel van de onder- en de bovengrens als een deelbereik van de integers gespecificeerd. De waarde die aan een deelbereik wordt toegekend moet vergeleken worden met de onder- en de bovengrens. Het is moeilijk de bereiken tijdens het compileren al te analyseren en daarom stellen de meeste implementaties deze controle uit tot tijdens de uitvoering van het programma. Het analyseren van deelbereiken komt op hetzelfde neer als het controleren of indices buiten hun grenzen komen. In Pascal is deze overeenkomst duidelijk omdat elk indextype een bereik is. Overflow en underflow bij het rekenen zijn bijzondere machine-afhankelijke voorbeelden van bereikfouten die meestal gemakkelijker ontdekt worden, omdat er speciale hardware voor is.

5.2 Controle tijdens compileren of tijdens uitvoeren van het programma

Typecontrole gaat na of de operaties gegevens van het juiste type te verwerken krijgen, dat wil zeggen typecontrole ontdekt typefouten. Het ligt het meest voor de hand te denken dat typecontrole tijdens het uitvoeren van het programma gebeurt, op het moment dat de gegevens aan de operaties worden aangeboden. Voordat een operatie wordt uitgevoerd, wordt dan de waarde van elk operand gecontroleerd. Maar run-time typecontrole heeft twee nadelen:

1. Het uitvoeren van het programma gaat langer duren.
2. In het algemeen is het niet mogelijk tijdens de uitvoering van een programma het type van een waarde te bepalen, vooral niet als de gegevens niet van een extra veld voorzien zijn waarin het type vermeld staat.

Daarom gebeurt typecontrole meestal tijdens het compileren, dus voordat het programma wordt uitgevoerd. Bij typecontrole tijdens het compileren wordt

gecontroleerd of het zeker is dat iedere operator in iedere expressie waarden van het juiste type te verwerken zal krijgen. Typecontrole tijdens compilatie controleert gewoonlijk alle operaties, aanroepen van procedures en functies, toekenningen en andere plaatsen waar waarden worden gebruikt. Meestal zet de compiler controles tussen de geproduceerde code, die typefouten ontdekken die tijdens de uitvoering van het programma kunnen optreden. Het volgende programma declareert twee variabelen; X is een integer en Z is een variabele die van ieder type kan zijn. We gebruiken het sleutelwoord IETS om aan te geven dat een variabele geen type heeft, dat er waarden van elk type aan kunnen worden toegekend. De procedure P heeft één integer parameter. Het programma roept P drie keer aan.

```
Y: INTEGER;
Z: IETS;                    -- een niet getypeerde variabele

P: procedure ( X: INTEGER ) is
begin
        ...
end P;

begin
        P(3);
        P(Y);
        P(Z);
        ...
end
```

Bij iedere aanroep moet het type van het argument overeenkomen met het type van de parameter. In de eerste aanroep is het argument 3 een integer constante en daarom behoeft de compiler geen controle in te bouwen. Het argument van de tweede aanroep is de variabele Y. De compiler ziet dat Y als integer is gedeclareerd en neemt aan dat de waarde van Y altijd een integer zal zijn. De compiler kan veilig van die veronderstelling uitgaan, omdat de compiler zelf elke toekenning aan de variabele Y controleert om te zien of aan Y alleen integer waarden worden toegekend. Het argument van de laatste aanroep is de variabele Z. Omdat Z een niet getypeerde variabele is, kent de compiler het type van de waarde die in Z is opgeslagen niet. Om typecontrole uit te voeren moet de compiler nu een controle inbouwen die tijdens het draaien van het programma wordt uitgevoerd. Het komt erop neer dat het type van het argument moet worden gecontroleerd voordat de procedure-aanroep wordt uitgevoerd. In de meeste talen is een declaratie als die van Z niet mogelijk, waardoor bijna elke typecontrole tijdens het compileren kan gebeuren.

Naast typecontrole tijdens compileren of uitvoeren van het programma bestaan er nog twee mogelijkheden. De vier mogelijkheden voor typecontrole zijn:

1. Controle tijdens het compileren.
2. Controle tijdens het linken.
3. Controle tijdens het uitvoeren.
4. Geen controle.

Het *linken* is de fase waarin apart gecompileerde programmasegmenten tot één executeerbaar geheel worden samengebracht. Stel bijvoorbeeld dat de bovenstaande procedure P afzonderlijk gecompileerd wordt van het segment:

```
P: procedure ( X: REAL ) is
begin
        ...
end P;
```

De typen van de twee procedures komen niet overeen, maar op welk moment kan dat gecontroleerd worden? Niet tijdens het compileren, want beide procedures worden apart gecompileerd. De compiler zou een controle kunnen inbouwen die tijdens de uitvoering van het programma wordt gedaan, maar dat verkleint de snelheid van het programma. Vele compilers voeren hier geen enkele controle uit. Maar het is ook mogelijk de typen te controleren tijdens het linken, als één geheel wordt gemaakt van de apart gecompileerde programma's.

De meeste talen en compilers hanteren een combinatie van deze vier alternatieven: tijdens het compileren, tijdens het uitvoeren, tijdens het linken en helemaal geen typecontrole. Typecontrole tijdens het linken komt maar zelden voor; daarom zullen we die mogelijkheid pas later in dit hoofdstuk behandelen. Als er in een bepaalde situatie geen typecontrole wordt uitgevoerd, zullen we dat een *type-lek* noemen. Een typelek betekent dat er typefouten zijn die onopgemerkt blijven. Sommige talen zijn zonder typecontrole ontworpen (bijvoorbeeld BCPL en Bliss). Vroeger was de wenselijkheid van dergelijke talen een punt van discussie, maar tegenwoordig zijn er maar weinigen die nog praten over de voordelen van het achterwege laten van typecontrole.

Bij de meeste programmeertalen worden de typen gecontroleerd tijdens het compileren of tijdens het uitvoeren van het programma. Typecontrole tijdens compilatie betekent dat de compiler kan vaststellen dat een waarde tijdens de uitvoering van het programma altijd op de juiste wijze gebruikt zal worden. Controle tijdens het uitvoeren houdt in dat de compiler code genereert voor

zo'n controle. Als er een typefout is, zal typecontrole tijdens compilatie ook tijdens het compileren een foutmelding of waarschuwing geven. Typecontrole tijdens de uitvoering van het programma zal alleen dan een foutmelding tijdens het uitvoeren geven als er werkelijk een typefout optreedt. Dit betekent dat typecontrole tijdens het compileren drie voordelen heeft, namelijk:

1. De foutmeldingen komen eerder.
2. Er wordt voor gezorgd dat bepaalde fouten tijdens de uitvoering van het programma niet voorkomen. Controle tijdens de uitvoering van het programma kan wel fouten ontdekken, maar ze niet voorkomen.
3. Programma's met extra code voor typecontrole zijn langzamer.

De bovenstaande observaties geven aan dat het te verkiezen is om typecontrole uit te voeren tijdens het compileren. Maar soms is het gemakkelijker controles in te bouwen die tijdens de uitvoering van het programma worden gedaan, dan te bewijzen dat een waarde op de juiste wijze wordt gebruikt bij de uitvoering. Controle op delen door nul en controle op deelbereiken in Pascal zijn daarvan goede voorbeelden. En wat belangrijker is, controle tijdens de uitvoering van het programma leidt tot een minder beperkte programmeeromgeving, wat soms zeer wenselijk is in sterk interactieve talen als APL en LISP. Deze talen hebben typeloze variabelen, waarin waarden van elk type kunnen worden opgeslagen. Een variabele kan dan bijvoorbeeld op het ene moment worden gebruikt voor het opslaan van een getal en op een ander moment voor een string. Deze talen worden soms *zwak getypeerde* of *typeloze* talen genoemd, niet omdat er geen typecontrole is, maar omdat de variabelen typeloos zijn. De waarden zijn wel getypeerd, en als een waarde tijdens de uitvoering van het programma onjuist wordt gebruikt, treedt er een typefout op. Deze typecontrole wordt mogelijk als aan elke waarde een *typeveld* wordt toegevoegd. Het typeveld specificeert het type van de waarde. Als de variabelen typeloos zijn, is het onmogelijk om de typecontrole tijdens het compileren uit te voeren, omdat het in het algemeen niet mogelijk is het type te bepalen van de waarde die in de variabele wordt opgeslagen.

De controverse tussen typeloze talen en getypeerde talen legt een belangrijk verschil bloot tussen de interactieve talen, die zich beter voor interpretatie lenen (met typecontrole tijdens de uitvoering van het programma), en de niet-interactieve talen, die eerder gecompileerd worden (met typecontrole tijdens de compilatie). Natuurlijk zijn er ook LISP-compilers en Pascal-interpretators, maar van huis uit leent LISP zich meer tot interpretatie en Pascal tot compilatie.

5.3 Een overzicht van typelekken

Een typefout is een fout in een bepaald programma. Een *typelek* is een fout in een programmeertaal. Een programmeertaal heeft *sterke typecontrole* als elke typecontrole tijdens het compileren plaatsvindt en er geen typelekken zijn. In zo'n taal is de gebruiker er zeker van dat alle typefouten tijdens de compilatie worden ontdekt. In vele talen komen veel typelekken voor. In deze paragraaf bespreken we enkele van de beruchtste typelekken.

In PL/I zijn pointers typeloos. Bij het gebruik van een pointer moet informatie worden gegeven over het object waarnaar de pointer wijst. Die informatie wordt door de programmeur gegeven. Maar de juistheid van die informatie wordt door de compiler of het run-time systeem niet nagegaan. Daardoor kan het voorkomen dat een pointer wijst naar een fixed binary getal, maar wordt gebruikt alsof hij naar een floating decimal getal wijst. Hier volgt een voorbeeld waarin twee floating decimal getallen bij elkaar worden opgeteld alsof het fixed binary getallen zijn:

```
X, Y, Z:  REAL;
J:        INTEGER;
P1, P2:   POINTER;

begin
        X  := 3.45;
        Y  := 4.56;

        P1 := ADDRESS(X);
        P2 := ADDRESS(Y);

        Z  := P1->J + P2->J;
        ...
```

De notatie `P1->J` betekent in PL/I dat het gegeven waarnaar de pointer `P1` wijst, moet worden geïnterpreteerd alsof het type gelijk is aan dat van `J`.

De variante records van Pascal hebben ook een typelek. Er is geen protectie voor de typediscriminant. Aan de typediscriminant kunnen waarden worden toegekend die volkomen onafhankelijk zijn van de waarde die aan de union is toegekend. De discriminant wordt misschien nooit gebruikt, zoals in onderstaand voorbeeld. Het is in Pascal mogelijk om met behulp van variante records twee floating-point getallen bij elkaar op te tellen alsof het integers zijn, zoals het volgende programma laat zien.

```
type U is record
            case SOORT: (I, R) is
            when I => INT_DEEL:  INTEGER;
            when R => REAL_DEEL: REAL;
            end case;
         end record;

X, Y, Z: U;

begin
        X.REAL_DEEL := 3.45;
        Y.REAL_DEEL := 4.56;
        Z.INT_DEEL  := X.INT_DEEL + Y.INT_DEEL;
end
```

Bij *onafhankelijke compilatie* kan het programma in stukken worden gedeeld; elk stuk kan dan onafhankelijk van de andere stukken worden gecompileerd. De gecompileerde stukken worden dan met de *linker* tot één objectmodule verenigd. Met de meeste compilers is onafhankelijke compilatie mogelijk, maar het komt maar zelden voor dat ook bij het linken of tijdens de uitvoering van het programma typecontrole plaatsvindt om na te gaan of de typen in de verschillende stukken wel consistent worden gebruikt. Beschouw de volgende twee programmasegmenten die onafhankelijk worden gecompileerd:

```
/* eerste segment */
F: function (A, B: INTEGER) return INTEGER is
begin
        return A+B;
end F;

/* tweede segment, apart gecompileerd */
F: function (A, B: REAL) return REAL is external;

X,Y,Z : REAL;

begin
        X := 3.45;
        Y := 4.56;
        Z := F(X, Y);
        ...
end
```

Als een functie wordt gedeclareerd als `external` (in Ada *separate*), betekent dat, dat de definitie van de functie voorkomt in een ander segment, dat apart wordt gecompileerd. Ada is zorgvuldig zo ontworpen dat typelekken worden vermeden, in het bijzonder tussen afzonderlijk gecompileerde stukken. Daardoor zou de compiler of de linker van Ada bij het bovenstaande programma een foutmelding geven. De meeste talen voeren zo'n controle niet uit en zouden de

twee reële getallen bij elkaar optellen alsof het integers waren. Dat soort fout is een *typelek tijdens het linken.* Andere typelekken tijdens het linken worden bijvoorbeeld gevormd door externe variabelen, die in het ene programmasegment van het ene type kunnen zijn en in het andere segment van een ander type.

Een ander klassiek typelek komt voor in talen als C en Pascal, waarin procedures als parameters kunnen worden doorgegeven. In sommige talen hangt het type van een procedure alleen af van het type van het resultaat, niet van het type van de argumenten. Het volgende programma telt twee reële getallen bij elkaar op alsof het integers zijn:

```
F: function (X,Y: INTEGER) return REAL is
begin
        return X+Y;
end F;

G: function (X,Y: REAL) return REAL is
begin
        return X+Y;
end G;

P: procedure (FUNC: function (...) return REAL) is
        X,Y,Z : REAL;
begin
        X := 3.45;
        Y := 4.56;
        Z := FUNC(X, Y);
end P;

begin                           -- hoofdprogramma
        P(G);
        P(F);
end
```

Merk op dat dit een ander probleem is dan bij typelekken tijdens het linken, omdat bovenstaand programma in zijn geheel wordt gecompileerd en er toch geen typefouten worden gemeld. De eerste aanroep van `P` bevat geen typefout. In de tweede aanroep van `P` is de functieparameter `F` inderdaad een `real` functie, zodat er geen typefout wordt gemeld. Maar als dan de functieparameter `F` binnen `P` wordt aangeroepen, treedt er een niet ontdekte typefout op. In moderne Pascal-compilers zijn deze typelekken weggewerkt door middel van veranderingen in de taal. De kopregel van de procedure kan bijvoorbeeld op de volgende manier veranderd worden, zodat het type van de operanden er ook in voorkomt:

```
P: procedure (FUNC: function (REAL, REAL) return REAL) is
```

Een *indexbereikfout* treedt op als een index van een array buiten de grenzen komt (dat wil zeggen niet tussen de onder- en de bovengrens van de array ligt). In de meeste talen wordt dat niet als een typefout beschouwd. De introductie van bereiken en deelbereiken was één van de vele vernieuwingen van Pascal. Arrays worden in Pascal gedefinieerd met behulp van deze bereiken en deelbereiken. Als bereiken en deelbereiken typen zijn, dan zijn bereikfouten typefouten; er doen zich dan bepaalde moeilijkheden voor. Ten eerste is het een onoplosbaar probleem om in het algemene geval tijdens het compileren vast te stellen of een programma bereikfouten bevat. In gevallen waarin de compiler niet kan vaststellen of een expressie in het juiste bereik valt, moet de compiler daarom een run-time controle invoegen. Dat komt op hetzelfde neer als het toevoegen van run-time controles voor fouten tegen indexbereiken. Zelfs een eenvoudige opdracht als

```
X := X+1;
```

moet van een run-time controle worden voorzien, want als het bereik van X loopt van i tot j, loopt het bereik van X+1 van i+1 tot j+1, en dat is geen deelbereik van i..j. Omdat run-time controles duur kunnen worden, kan het invoegen van zulke run-time controles in sommige compilers worden afgezet. Die truc wordt zo algemeen toegepast dat het volgende implementatie-afhankelijke typelek in vele talen kan worden aangetroffen. Als men weet hoe er geheugen wordt gealloceerd is het mogelijk met een programma als het volgende twee reele getallen bij elkaar op te tellen alsof het integers zijn:

```
A: array (1..4) of INTEGER;
X,Y,Z: REAL;

begin
        X := 3.45;
        Y := 4.56;
        Z := A(5)+A(6);
end;
```

Als het geheugen wordt gealloceerd in de volgorde van de declaraties, dan hebben A(5) en X hetzelfde adres, en A(6) en Y ook*. Zo ja, dan worden met A(5)+A(6) de reals X en Y bij elkaar opgeteld alsof het integers zijn.

---

* Dit geldt als integers en reals evenveel plaats innemen. In sommige implementaties nemen reals twee keer zoveel plaats in als integers. In dat geval hebben A(7) en Y hetzelfde adres.

Een klassiek probleem met pointers is het verkeerde gebruik van de nulpointer. De nulpointer wordt voor verschillende doeleinden gebruikt, zoals voor het beëindigen van een geketende lijst. Het is een goede gewoonte om eerst na te gaan of een pointer niet nul is, alvorens de pointer te gebruiken. Als dat niet wordt gecontroleerd, zijn de gevolgen onvoorspelbaar en kunnen ze tot moeilijk vindbare fouten leiden. Eén van de manieren om dit probleem te vermijden is het schrappen van de waarde nul voor pointers. In situatie waarin een pointer een nulwaarde moet hebben, kan gebruik gemaakt worden van een type als:

```
type U is union (VOID, pointer P);
```

Bij het gebruiken van een waarde van het type U moet altijd eerst worden gecontroleerd of de waarde het type VOID heeft. Met deze methode wordt een lelijke run-time fout vervangen door een netjes gecontroleerd gegevenstype. Helaas zijn niet veel programmeurs bereid een gemakkelijke notatie op te geven voor veiligheid.

Typelekken in een taal hebben ook bepaalde voordelen; ze kunnen af en toe nuttig zijn. In de taal C is het idee van het typelek geformaliseerd met de constructie *cast*. Met cast is het mogelijk een gegeven van het ene type te laten interpreteren als een gegeven van een ander type. Behalve in een enkel geval, waarin de cast-operatie de representatie van een waarde echt verandert (voornamelijk tussen numerieke typen), verandert de cast-operator alleen de typeaanduiding van de waarde, zonder de representatie zelf te veranderen. Deze aanpak maakt de flexibiliteit van typeloze pointers mogelijk in een omgeving waarin pointers getypeerd plegen te zijn.

5.4 Operatoridentificatie en overloading

De fundamentele manier, en volgens sommigen de enige, om expressies op te bouwen is met behulp van operatoren en operanden. Een operatie definieert een functie op de waarde van de operanden. Om de operatie aan te geven wordt een operatorsymbool gebruikt. Als elk operatorsymbool een unieke operatie definieert, dan is de typecontrole eenvoudig (afgezien van impliciete conversies). Elk operatorsymbool zou dan het type van de operanden volledig bepalen. De operator zou ook het type van het resultaat bepalen.

Gewoonlijk wordt een operatorsymbool voor meer dan één operatie gebruikt. In ALGOL 68 staat de operator '+' voor optelling van reals, optelling van inte-

gers, concatenatie van strings en eventuele operaties die de programmeur zelf heeft gedefinieerd. Dit proces wordt *overloading* genoemd. De compiler moet bepalen welke operatie met een overloaded operatorsymbool wordt bedoeld. Dat wordt *operator-identificatie* genoemd. Als de taal overloading van symbolen door de gebruiker niet toestaat, kan de compiler worden uitgerust met een gespecialiseerde routine voor identificatie van de overloaded operatoren van de taal. Maar als de taal de gebruiker toestaat in een of andere vorm zelf overloaded operatoren te definiëren, dan moet de routine voor operatoridentificatie op een algemene manier worden beschreven. In ALGOL 68 en Ada mag de programmeur zelf overloaded operatoren definiëren.

In ALGOL 68 kan bij één enkel operatorsymbool elk gewenst aantal functies behoren, zolang aan bepaalde voorwaarden wordt voldaan. Die voorwaarden zorgen ervoor dat ondubbelzinnig kan worden bepaald welke operator bedoeld wordt. Als *A* en *B* twee van de functies zijn die bij een bepaald operatorsymbool horen, dan mogen de typen van de overeenkomstige parameters van *A* en *B* niet *firmly related* zijn, wat in de terminologie van ALGOL 68 wil zeggen dat de typen niet in elkaar kunnen worden overgevoerd in wat een *firm context* wordt genoemd.

De routine voor operatoridentificatie werkt voor ALGOL 68 volledig bottom-up. Gegeven de typen van de operanden is er in de lijst van beschikbare operaties altijd ten hoogste één mogelijke operatie die klopt. (Zodra er meer dan één klopt, moet de compiler een foutmelding geven voor de nieuwe operator.) De routine voor operatoridentificatie is dus een eenvoudige zoekprocedure, die rekening moet houden met type-equivalentie en met impliciete conversies. Als er geen operatie wordt gevonden die klopt met het type van de operanden, treedt er een fout op, die kan worden aangemerkt als een fout in het type van een operand of als een ongedefinieerde operator.

Anders dan ALGOL 68 kent Ada geen impliciete conversie bij operaties en aanroepen van procedures. Verder gebruikt Ada de ‘naamequivalentie’ in plaats van de ‘structurele equivalentie’ (paragraaf 5.6). Beide eigenschappen dragen bij tot een eenvoudiger mechanisme voor overloading in Ada. Maar er zijn twee andere verschillen die overloading in Ada ingewikkelder maken dan in ALGOL 68. Ten eerste laat Ada behalve voor operatorsymbolen ook overloading toe voor namen van procedures en constanten van opgesomde typen. En op de tweede plaats staat Ada overloading toe op elke plaats waar dat niet tot dubbelzinnigheid leidt. Terwijl de algoritme voor operatoridentificatie in ALGOL 68 volkomen bottom-up verloopt, is het in Ada nodig zowel bottom-up

als top-down te werk te gaan voor het identificeren van de operatoren. Dit verschil wordt meteen duidelijk als we kijken naar overloading bij constanten van een opgesomd type. Een constante is een operator met nul operanden en daarom zijn er geen operanden die kunnen worden gebruikt om de dubbelzinnigheid van een overloaded constante op te lossen. De dubbelzinnigheid moet bij overloaded constanten worden opgelost met behulp van de context en daarom is er een top-down analyse nodig. Ook voor operatoren met operanden kan een top-down analyse noodzakelijk zijn. Laten we eens kijken naar de volgende overloaded functies in Ada:

```
F: function ( X: INTEGER) return INTEGER;
F: function ( X: INTEGER) return REAL;

X: INTEGER := F(5);
```

De functie `F` is overloaded en de dubbelzinnigheid kan niet op grond van alleen het type van de parameter worden opgelost. Alleen al om die reden is dit tweetal functies in ALGOL 68 niet toegestaan. Een Ada-compiler moet de dubbelzinnigheid oplossen met behulp van de context. Op basis van het feit dat het resultaat van de functie wordt toegekend aan een integer variabele, zal de compiler de eerste functie kiezen. Een Adacompiler moet ook vaststellen of een bepaalde overloaded expressie intrinsiek dubbelzinnig is. In het volgende voorbeeld kan de dubbelzinnigheid niet worden opgelost, zodat het geen correct Ada is.

```
type MUNT is (POND, DOLLAR, GULDEN);
type GEWICHT is (POND, ONS, KILO);
P: procedure ( X: MUNT);
P: procedure ( X: GEWICHT);

P(POND)
```

Ada staat overloading niet voor alle identifiers toe, maar laten we uitgaan van een taal waarin overloading wel voor alle identifiers is toegestaan. Als bijvoorbeeld overloading van variabelen is toegestaan, dan kunnen we hebben:

```
X, Y: INTEGER;
X, Z: (ROOD, GEEL, BLAUW);
Y: BOOLEAN;
Z: (ROOD, WIT, BLAUW);
```

Elke identifier staat hier voor twee variabelen. We moeten in staat zijn bij elke toepassing op ondubbelzinnige wijze vast te stellen om welke variabele het

gaat. In onderstaande opdrachten kan X alleen worden opgevat als de integer variabele, anders passeren de opdrachten de typecontrole niet.

```
X := 5;
X := X+Y;
if X=Y then ...
if Y then ...
```

In de laatste regel moet Y wel slaan op de Boolean variabele. Maar niet alle gevallen zijn ondubbelzinnig:

```
X := SUCC(X);
PRINT(X);
if X=X then ...
X := ROOD;
```

De derde opdracht heeft de interessante eigenschap dat die syntactisch dubbelzinnig is, maar semantisch ondubbelzinnig. De opdracht is syntactisch dubbelzinnig omdat de identifier X zowel voor de integer variabele als voor de kleurvariabele kan staan. De opdracht is semantisch ondubbelzinnig omdat in beide gevallen de gelijkheidsoperator true zal afleveren.

Het is duidelijk dat overloading zonder beperkingen ongewenst is. Men zou zelfs het standpunt kunnen innemen dat elk soort overloading ongewenst is, zelfs voor constanten van opgesomde typen. Maar voor de namen van operatoren en invoer/uitvoerroutines is overloading met veel succes toegepast. We moeten afwachten in welke mate overloading in toekomstige talen zal worden gebruikt.

5.5 Impliciete conversies

Er zijn vele redenen om gegevens van het ene type naar het andere te converteren. Integer waarden worden soms gebruikt als real waarden. Soms is het handig de numerieke waarde van een teken te kennen. Zulke conversies kunnen meestal worden uitgevoerd met behulp van een conversieroutine. Om in ALGOL 68 de real C expliciet naar een integer waarde te converteren, kan men één van de volgende expressies gebruiken:

```
entier(C)
round(C)
```

De eerste routine verwijdert het deel achter de decimale punt; dat wordt *afkappen* genoemd. De tweede functie telt eerst .5 bij het operand op en verwijdert dan het deel achter de decimale punt. De eerste functie converteert het reële getal .75 naar 0, de tweede naar 1. In Ada schrijft men voor het expliciet converteren van een integer naar een real:

```
REAL(C)
```

Een *impliciete conversie* of *coërcie* is een conversie die de compiler automatisch in de code invoegt. De programmeur geeft niet speciaal aan dat de conversie moet plaatsvinden. In plaats daarvan neemt de compiler aan dat de programmeur wil dat de conversie plaatsvindt. Een heel gebruikelijke impliciete conversie is die van integer naar real. In de meeste talen is de volgende toekenningsopdracht toegestaan en zal de compiler een conversieroutine toevoegen die de integer waarde converteert naar een real waarde.

```
X: REAL;
J: INTEGER;

X := J;
```

In verschillende talen zijn verschillende soorten coërcies toegestaan. De conversie van integer naar real wordt als veilig beschouwd, omdat de waarde van het getal niet wordt veranderd. Als een taal dus conversies toestaat, zullen coërcies van integer naar real daar waarschijnlijk bij zijn. Coërcies van real naar integer worden om twee redenen niet als veilig beschouwd. Op de eerste plaats kan de betekenis van een getal drastisch veranderen. Als .25 naar een integer wordt geconverteerd, is het resultaat 0, een groot verschil met een kwart. Deze conversie verandert niet alleen het type, maar ook de waarde. Kijk maar eens wat een verschil het maakt als het getal de deler is in een deling. Een tweede probleem is dat er, zoals we eerder hebben gezien, verschillende conversieroutines zijn die kunnen worden gebruikt om een real naar een integer te converteren. Deze problemen leiden tot de gedachte dat conversies van real naar integer beter expliciet kunnen zijn dan impliciet.

Coërcies bemoeilijken het proces van typecontrole. Als bijvoorbeeld coërcie van integer naar real toegestaan is, dan moet op plaatsen waar een real waarde wordt verwacht, een integer waarde ook aanvaardbaar zijn. Die situatie kan tot dubbelzinnigheid leiden als er ook overloading van operatoren optreedt. Beschouw de volgende overloaded operator in het licht van de coërcie van integer naar real:

```
function O ( X: REAL; Y: INTEGER) is ...
function O ( X: INTEGER; Y: REAL) is ...
```

Welke functie moet er worden gebruikt als beide operanden integers zijn. In ALGOL 68 worden zulke dubbelzinnigheden vermeden doordat coërcies alleen mogen plaatsvinden in een bepaalde context. In bovenstaand voorbeeld kunnen integers niet impliciet naar reals worden geconverteerd. In PL/I is voor een andere oplossing van dit probleem gekozen. PL/I kent speciale regels voor de volgorde waarin coërcies worden uitgevoerd, zodat maar één bepaalde interpretatie wordt gekozen, ook als er vele interpretaties mogelijk zijn.

Hoe meer coërcies een programmeertaal heeft, des te minder waardevol wordt typecontrole. Potentiële typefouten worden gemakkelijk coërcies en blijven onontdekt. Coërcies kunnen worden opgevat als één van de manieren waarop een compiler fouten in het programma kan vinden en 'verbeteren'. Maar in het algemeen worden coërcies niet gemeld, waardoor het nut van typecontrole achterwege blijft. Het beste voorbeeld van coërcies die tot het uiterste zijn doorgevoerd leveren de numerieke conversies van PL/I. In PL/I kan elk numeriek type naar elk ander numeriek type worden geconverteerd -- ook bits, bitstrings en tekenstrings. Daardoor kunnen er rare dingen gebeuren. Geen van de volgende stukjes PL/I geven een compilerfout, maar goede compilers geven wel een waarschuwing.

```
IF 1<X<5 THEN CALL Q;

DO J=1, J=2;

DO J=1 TO 2 BY .1;

A = 20 + 2/3;
```

De eerste opdracht roept `Q` altijd aan, omdat `1<X<5` altijd true oplevert. De deelexpressie `1<X` levert true of false, hetgeen naar 0 of 1 wordt geconverteerd; omdat 5 groter is dan 0 en groter dan 1, levert de expressie als geheel true als resultaat. De tweede opdracht voert de lus twee maal uit, beide keren met 1 als waarde voor `J`. De eerste `=` is een toekenning, maar de tweede `=` is een vergelijking. De expressie `J=2` levert false, dat naar 1 wordt geconverteerd. De derde opdracht leidt tot een oneindige herhaling, omdat de floating decimal constante .1 wordt geconverteerd naar de fixed binary constante 0. De laatste opdracht geeft `A` de waarde 2/3. De expressie `2/3` levert een fixed decimal getal met precisie (6,6) en als dat wordt opgeteld bij een fixed decimal constante 20 met

precisie (2,0), dan heeft het resultaat, volgens de conversieregels van PL/I, de precisie (6,6), zodat er aan de linkerkant twee cijfers worden afgekapt.

Het is de moeite waard de coërcies van ALGOL 68 te bestuderen, deels omdat die met zorg zijn gekozen, maar vooral om coërcies te zien die niet onmiddellijk voor de hand liggen. ALGOL 68 kent zes coërcies. Ze heten widening, dereferencing, deproceduring, uniting, rowing en voiding. *Widening* is de coërcie waaronder conversie van integer naar real en van real naar complex vallen. *Dereferencing* is de coërcie die het type *reference to T* converteert naar het type *T*. Dit is een heel gebruikelijke coërcie, die in de meeste programmeertalen voorkomt. In vele talen wordt van deze coërcie stilzwijgend uitgegaan, zo sterk zelfs dat de taaldefinitie het bestaan ervan niet eens noemt. Bij een expressie als X+Y zijn twee dereferencing coërcies betrokken, die de variabelen X en Y converteren naar integer waarden. Bliss kent deze coërcie niet; dereferencing wordt daar aangegeven met de unaire punt-operator. Om de inhoud van X en Y bij elkaar op te tellen moet men schrijven:

```
.X+.Y
```

Met de expressie X+Y worden in Bliss de adressen van X en Y bij elkaar opgeteld.

Deproceduring is de coërcie die een waarde van het type *procedure zonder parameters met een resultaat van type T* converteert naar het type *T*. Deproceduring komt neer op het aanroepen van de procedure. Het resultaat van de coërcie is de waarde die de procedure oplevert. Deproceduring maakt het gewoon mogelijk een procedure aan te roepen zonder haakjes te gebruiken. Om P, een procedure met parameters, aan te roepen, moet men de argumenten meegeven. Als P geen argumenten heeft, komt er niets tussen de haakjes na P. In sommige talen zijn die haakjes dan toch verplicht, zodat men moet schrijven:

```
P()
```

In andere talen kan volstaan worden met:

```
P
```

In dit laatste geval bestaat er een mogelijke dubbelzinnigheid. Moet het resultaat van de expressie P de procedure P zijn of de waarde die het resultaat is van een aanroep van P? Het antwoord kan uit de context worden afgeleid. In ALGOL 68 dicteert de context of de deproceduring coërcie moet plaatsvinden.

Uniting converteert een waarde van het type *T* naar een waarde van het type *union van T en X en Y ....* Deze coërcie is nuttig als er algemene routines worden aangeroepen of er toekenningen aan union-variabelen plaatsvinden. Bij voorbeeld:

```
X: union (INTEGER, REAL);
X := 5;

P: procedure (A, B: union (INTEGER, REAL)) is
begin
        ...
end P;

P(3, 4.5);
```

In de toekenningsopdracht staat aan de rechterkant een integer waarde, maar aan de linkerkant een variabele van een uniontype. De integer waarde wordt met behulp van uniting impliciet geconverteerd naar het type van X. Deze uniting coërcie verandert de waarde niet. In de aanroep van P in de laatste opdracht zijn beide argumenten van het verkeerde type; ze moeten allebei met behulp van uniting naar het type van de parameters worden geconverteerd.

Rowing is een coërcie van ALGOL 68 die een waarde van type *T* converteert naar een waarde van type *row of T*. Een *row* van ALGOL 68 is gewoon een array. Deze coërcie genereert een array van één element. Rowing is nuttig voor arrays met veranderlijke lengte (in ALGOL 68 *flexible rows* genoemd), maar voor arrays met vaste lengte is het nut gering. Het komt vaak voor dat men een array met veranderlijke lengte een startwaarde wil geven bestaande uit een enkel element. In ALGOL 68 is er wel een eenvoudige manier om een array van nul of van twee elementen toe te kennen aan een variabele, bijvoorbeeld:

```
X: array () of INTEGER;
X := ();
X := (2,3);
```

waarin X een integer array van veranderlijke lengte is. Maar er is geen syntactisch gemakkelijke manier om een array met één element toe te kennen, omdat haakjes ook worden gebruikt om de volgorde van uitwerking in een expressie aan te geven. Om dit probleempje op te lossen is de coërcie rowing ingevoerd, die de volgende toekenning mogelijk maakt:

```
X := 4;
```

De laatste coërcie van ALGOL 68 is *voiding*. Net als van dereferencing gaan vele talen ook stilzwijgend van voiding uit. In ALGOL 68 is voiding nodig om een onderscheid te maken tussen procedures en functies. In ALGOL 68 moet elke procedure een waarde van een of ander type opleveren. Het type VOID wordt gebruikt om aan te geven dat het niet geeft wat het resultaat is, omdat het toch verder niet wordt gebruikt. Beschouw onderstaande procedure P:

```
P: procedure  return VOID is
begin
        Y:=3
end P;
```

In ALGOL 68 is het type van het resultaat van een procedure gelijk aan het type van de laatste expressie van de procedure. In het laatste voorbeeld rijst dan een klein probleempje, doordat het type van de laatste expressie integer is, terwijl het VOID zou moeten zijn. Dit probleempje wordt in ALGOL 68 opgelost met de coërcie voiding, die elk type *T* converteert naar het type VOID.

5.6 Type-equivalentie

Iets dat elke algoritme voor typecontrole moet doen is bepalen of twee typen hetzelfde zijn. Dit lijkt op het eerste gezicht een simpele taak; maar, zoals in het volgende voorbeeld zal blijken, zijn er vragen die op verschillende manieren kunnen worden beantwoord. Zijn in het volgende programmasegment de typen ZWART en WIT gelijk?

```
type ZWART is INTEGER;
type WIT is INTEGER;

Z: ZWART;
W: WIT;
I: INTEGER;

begin
        W := 5;
        Z := W;
        I := Z+3;
        ...
```

Is ZWART een andere naam voor INTEGER? Zijn ZWART en INTEGER namen voor hetzelfde type of zijn het namen voor verschillende, maar op elkaar lijkende typen? Als ze hetzelfde zijn, dan zijn al deze toekenningsopdrachten correct. Als ze verschillend zijn, welke toekenningen zijn dan correct (aangenomen dat

er geen coërcies zijn toegestaan)? En als de typen verschillend zijn, is 5 dan een waarde van beide typen? Dit zijn enkele van de vragen die door een simpele situatie als deze worden opgeroepen.

De vraag wanneer twee typen als gelijk moeten worden beschouwd, wordt het *type-equivalentieprobleem* genoemd. Er zijn twee manieren om dit probleem aan te pakken, genaamd *naam-equivalentie* en *structurele equivalentie*. Naam-equivalentie komt erop neer dat typen alleen gelijk zijn als ze dezelfde naam dragen. Structurele equivalentie komt erop neer dat typen gelijk zijn als er dezelfde structuur aan ten grondslag ligt. Er zijn ook andere manieren om het type-equivalentieprobleem aan te pakken, die tussen deze beide methoden in liggen; maar we zullen hier alleen de principes van naam- en structurele equivalentie bestuderen.

Het gebruik van structurele equivalentie roept enkele problemen op. Beschouw de volgende typen:

```
type A is record
               X: INTEGER;
               N: pointer A;
          end record;

type B is record
               X: INTEGER;
               N: pointer A;
          end record;

type C is record
               X: INTEGER;
               N: pointer record
                               X: INTEGER;
                               N: pointer C;
                          end record;
          end record;

type D is record
               Y: INTEGER;
               N: pointer D;
          end record;
```

Alle bovenstaande typen zijn structureel gelijk. Elk type bestaat uit een record met twee componenten, een integer en een pointer naar dezelfde soort record. ALGOL 68 gaat uit van structurele equivalentie en beschouwt de typen A, B en C als verschillende spellingen voor hetzelfde type. Maar ALGOL 68 beschouwt type D als een ander type, omdat de namen van de componenten van een record

als deel van het type worden beschouwd. Andere complicaties in de aanpak van type-equivalentie in ALGOL 68 zijn onder andere de unions, waarin de volgorde van de typen er niet toe doet. Dat betekent dat

```
union (INTEGER, REAL)
```

equivalent is met

```
union (REAL, INTEGER)
```

De methode van type-equivalentie die in Pascal wordt toegepast, wordt uit de oorspronkelijke definitie niet duidelijk, maar de gangbare interpretatie lijkt tegenwoordig naamequivalentie te zijn. In Ada wordt naamequivalentie officieel aangegeven. Bij die methode worden twee verschillende typenamen altijd geacht verschillende typen aan te duiden, ook al hebben ze dezelfde structuur. Het probleem dat rijst bij het gebruik van constanten en operatoren voor de typen `ZWART` en `WIT` wordt in Ada opgelost met behulp van het concept *afgeleide typen*. Een afgeleid type wordt aangegeven met het sleutelwoord `new`, zoals in de declaratie:

```
type GROEN is new INTEGER;
```

Het type `GROEN` is een van `INTEGER` afgeleid type en de kenmerken van het type `GROEN` worden ook van die van `INTEGER` afgeleid. Tot deze kenmerken behoren alle constanten en functies die op integers zijn gedefinieerd.

Een klein probleem doet zich bij naamequivalentie voor als er een type-expressie wordt gebruikt die geen naam krijgt. In de volgende declaraties zijn de typen structureel gelijk, maar het is niet duidelijk of ze bij de methode van naamequivalentie gelijk zijn:

```
A:    record
         X, Y: INTEGER;
      end record;

B,C: record
         X, Y: INTEGER;
      end record;
```

De typen van `A`, `B` en `C` worden *anonieme typen* genoemd, omdat ze geen naam hebben. Naamequivalentie is hier duidelijk niet van toepassing. Hier volgen een paar mogelijke interpretaties.

1. Een anoniem type is met geen enkel ander type equivalent. Dat zou betekenen dat A, B en C van drie verschillende typen zijn.
2. Bij een anoniem type fungeert de type-expressie als de naam van het type. Dat zou betekenen dat A, B en C van hetzelfde type zijn.
3. Elke introductie van een anoniem type creëert een nieuw type. Dat betekent dat B en C van hetzelfde type zijn, dat verschilt van het type van A.
4. Anonieme typen zijn niet toegestaan, in welk geval de bovenstaande situatie niet kan voorkomen.

In Ada wordt de derde interpretatie gebruikt (behalve voor subtypen).

5.7 Onafhankelijke compilatie

Er zijn vele goede redenen voor aparte compilatie, maar aparte compilatie roept een aantal problemen op voor de typecontrole. De interface tussen twee onafhankelijk gecompileerde modulen omvat in het algemeen procedures en gegevens. Als er geen typecontrole wordt uitgevoerd op die interface, dan is er sprake van een typelek. Er zijn verschillende manieren om dat typelek te dichten.

Het is mogelijk typecontrole tijdens de uitvoering van het programma te laten plaatsvinden. Alle externe namen kunnen worden voorzien van een typeveld en er kan run-time code worden toegevoegd om na te gaan of de typen van parameters en argumenten overeenkomen. Externe variabelen moeten vóór elk gebruik worden gecontroleerd. Helaas gaan procedureaanroepen op deze manier veel langer duren.

Het is ook mogelijk typecontrole te laten plaatsvinden tijdens het linken. Als twee modulen aan elkaar gekoppeld worden kunnen de interfaces gecontroleerd worden. Deze oplossing vereist een hoge mate van samenwerking tussen de compiler en de linker. De compiler moet van elke externe variabele het type genereren. De linker moet nagaan of de typen gelijk zijn. Het is daarbij niet voldoende alleen de namen te controleren, zelfs als de taal uitgaat van naamequivalentie. Twee modulen kunnen dezelfde typenaam gebruiken, maar voor verschillende representaties. Daarom kan de compiler er niet mee volstaan de typenamen door te geven, maar moet hij ook de structuur aangeven die bij elke typenaam hoort. De linker wordt dan ingewikkelder, om niet alleen de namen te kunnen vergelijken, maar ook de erbij behorende typen. Het voordeel van controle tijdens het linken is dat het uitvoeren van het programma niet langer gaat duren.

Opgaven

1. Deelbereiken worden gewoonlijk niet als een type beschouwd. Neem aan dat deelbereiken wel typen zijn en geef de fouten aan in het volgende voorbeeld. Neem ook aan dat impliciete conversies zijn toegestaan van het ene deelbereik naar het andere als het eerste deelbereik bevat is in het tweede.

```
B1_VAR: 3..9;
B2_VAR: 4..10;
begin
        B2_VAR := 3;
        B2_VAR := B1_VAR;
        B2_VAR := B2_VAR + 1;
        B2_VAR := B1_VAR + 1;
        if B1_VAR≠3 then
                B2_VAR := B1_VAR;
        ...
```

   Bij een bepaalde operatie kan het zijn dat sommige combinaties van waarden van de operanden een typefout opleveren. Als alle combinaties een typefout opleveren, dan behoort de expressie tijdens het compileren als een typefout te worden aangemerkt. Als geen enkele combinatie een typefout oplevert behoort de expressie nooit als een typefout te worden aangemerkt, noch tijdens het compileren, noch tijdens het uitvoeren van het programma. Maar stel eens dat alleen bepaalde, maar niet alle combinaties een typefout opleveren, zoals in de voorbeelden hierboven. Moet zo'n expressie dan als typefout worden gemeld tijdens het compileren, ook al is het programma zo geschreven, dat zo'n onjuiste combinatie tijdens het uitvoeren van het programma nooit optreedt? Welke implementatieproblemen doen zich voor als zulke expressies als typefouten worden beschouwd die tijdens de compilatie moeten worden gesignaleerd?

2. Beschouw het volgende voorbeeld van overloading. Geef bij elk voorkomen van '+' aan welke operatie bedoeld is of geef, in geval van dubbelzinnigheid, alle mogelijke interpretaties. Neem aan dat '+' al gedefinieerd is als integer optelling.

```
function "+" (X:INTEGER; Y:LIST)    return LIST;
function "+" (X:LIST; Y:INTEGER)    return LIST;
function "+" (X:LIST; Y:LIST)       return LIST;

J,K: INTEGER;
A,B: LIST;
```

```
begin
     K:=J+J;
     A:=A+J;
     B:=J+K+A;
     A:=J+A+K;
     B:=(3+A)+(B+5);
end
```

3. Ga na wanneer in uw favoriete programmeertaal (bijvoorbeeld Pascal) en in uw favoriete compiler de typecontrole plaatsvindt. Controleert de compiler de bereiken? Welke typelekken komen er voor? Welke typen zijn equivalent? Let op details als parameteroverdracht, toekenning, bereiken en pointers. Test de mogelijkheden tot onafhankelijke compilatie van de compiler. Is de typecontrole voor externe variabelen en voor parameters waterdicht? Hoe en wanneer worden ze gecontroleerd? Waarnaar wijst de nulpointer in de implementatie? Wat zijn de regels voor typecontrole?

4. Aan het eind van paragraaf 5.7 worden vier interpretaties opgesomd. Welke zou de lezer kiezen indien hij of zij zelf een programmeertaal zou ontwerpen met naamequivalentie, en waarom?

Literatuur

Een goede studie van de typelekken in Pascal is te vinden in Welsh et al. (1977) en in een bundel artikelen in paragraaf 5 van Wasserman (1980). Tennent (1978) en Berry en Schwartz (1979) gaan verder in op het type-equivalentie-probleem. Eggert (1981) geeft een versie van Pascal waarin alle fouten tijdens het compileren ontdekt kunnen worden. In *SIGPLAN Notices* en in conferenties over de taal Ada (zie Baker, 1982) zijn een aantal artikelen verschenen over operatoridentificatie in Ada. Gannon (1977) rapporteert over een proefondervindelijke studie van typeloze en getypeerde talen.

Het tijdens het compileren bepalen van typen voor programma's zonder declaraties is voor verscheidene talen diepgaand bestudeerd, zoals voor ML door Milner (1978), voor APL door Miller (1979), voor Smalltalk door Suzuki (1981), voor SETL door Tenenbaum (1974), voor ABC (voorheen B) door Meertens (1983) en in het algemeen door Cousot en Cousot (1977) en Kaplan en Ullman (1980). Zie ook hoofdstuk 10.

6

Voorbeelden van typecontrole

6.1 Inductieve definities en typecontrole

Uitgaande van alle informatie uit de vorige paragrafen moet de ontwerper van een compiler een algoritme voor typecontrole in elkaar zetten waarin hij alle beslissingen uit het taalontwerp implementeert die met typecontrole te maken hebben. De complexiteit van algoritmen voor typecontrole varieert afhankelijk van het samenspel van operatoridentificatie, coërcies en de aanpak van type-equivalentie. Ook de *context* kan in belangrijke mate de complexiteit van een algoritme voor typecontrole bepalen. Dat wil zeggen dat de omringende situatie kan bepalen of een bepaalde regel moet worden toegepast. Bij een expressie bijvoorbeeld kan het voor de typecontrole nodig zijn de declaraties te kennen van de variabelen die in de expressie worden gebruikt. Alvorens over te gaan tot het ontwerpen van algoritmen, is het noodzakelijk een goed begrip te hebben van de complexiteit van typecontrole. In deze paragraaf geven we inductieve definities voor het beschrijven van regels voor typecontrole.

Expressies in programmeertalen worden recursief gedefinieerd; op dezelfde manier kunnen eigenschappen of kenmerken van expressies recursief worden gedefinieerd. Voordat we inductieve definities gaan toepassen op typecontrole, beschrijven we eerst een eenvoudige taal en een eenvoudige inductieve definitie.

Beschouw de volgende taal M:

Syntaxis van de taal

E ::= (zij L gelijk E in E) | EE | L
L ::= *a* | *b* | *c* | *d* | *e*

We zouden graag bepaalde functies op deze taal willen definiëren. We willen misschien de lengte van strings in de taal definiëren. Zij *Lengte* een functie van strings in M naar de gehele getallen, die het aantal tekens in de expressie aflevert die ongelijk zijn aan een spatie. Bij het toepassen van een functie gebruiken we in plaats van de gebruikelijke ronde haakjes vierkante haken om syntactische expressies te begrenzen. We willen bijvoorbeeld graag schrijven:

Lengte[(zij *a* gelijk *cc* in *aabaa*)] = 21

We kunnen de volgende inductieve methode gebruiken om *Lengte* formeel te definiëren.

Lengte[(zij L gelijk E_1 in E_2)] = 14 + Lengte[E_1] + Lengte[E_2]
Lengte[E_1 E_2] = Lengte[E_1] + Lengte[E_2]
Lengte[L] = 1

Merk op dat we niet-terminale symbolen van een index moeten voorzien, opdat er geen dubbelzinnigheid ontstaat bij het toepassen van deze definitie*. Om de definitie toe te passen op de expressie

(zij *a* gelijk *cc* in *aabaa*)

moeten we eerst bepalen welke regels van de grammatica gebruikt zijn om de expressie voort te brengen. Omdat het eerste alternatief van de grammatica is gebruikt, kunnen we zeggen dat

Lengte[(zij *a* gelijk *cc* in *aabaa*)] = 14 + *Lengte*[*cc*] + *Lengte*[*aabaa*]

---

* Dubbelzinnige grammatica's kunnen inductieve definities ook dubbelzinnig maken. Hoewel we voor de taal M een dubbelzinnige grammatica hebben gebruikt, zijn bovenstaande definities niet dubbelzinnig.

De lengte van *cc* en *aabaa* kan op dezelfde manier worden bepaald (door het tweede alternatief te gebruiken):

Lengte[*cc*] = *Lengte*[*c*] + *Lengte*[*c*]

Ten slotte kunnen we met behulp van het laatste alternatief de berekening afmaken:

Lengte[*cc*] = *Lengte*[*c*] + *Lengte*[*c*] = 1 + 1 = 2
Lengte[(zij *a* gelijk *cc* in *aabaa*)] = 14 + *Lengte*[*cc*] + *Lengte*[*aabaa*]
= 14 + 2 + 5

Inductieve definities zoals de bovenstaande kunnen voor vele van dit soort doeleinden worden gebruikt, ook om de semantiek van een taal te definiëren. Op dat onderwerp komen we in hoofdstuk 12 en 13 terug.

In dit hoofdstuk gebruiken we inductieve definities om typecontrole te definiëren. Om het proces te illustreren zullen we een deelverzameling van expressies en typen definiëren die in de meeste talen voorkomen. Vervolgens definiëren we *Type* als een functie van expressies naar typen.

Syntaxis van expressies

E ::= N | I | E(E) | E+E
N ::= 0 | 1 | 2 | ... | 32767
I ::= *a* | *b* | *c* | ... | *z*

We nemen aan dat elke variabele is gedeclareerd met een type uit de volgende grammatica:

Syntaxis van typen

T ::= int | array-of-T | function(T)-returns-T

De *symbooltabel* SymTab is een functie van identifiers (I) naar typen (T). Dus als “a” is gedeclareerd als “array-of-int”, dan is SymTab[a] = array-of-int.

We kunnen nu het volgende voorbeeld geven van een inductieve definitie van het type van een expressie:

Type[N] = int

Type[I] = SymTab[I]

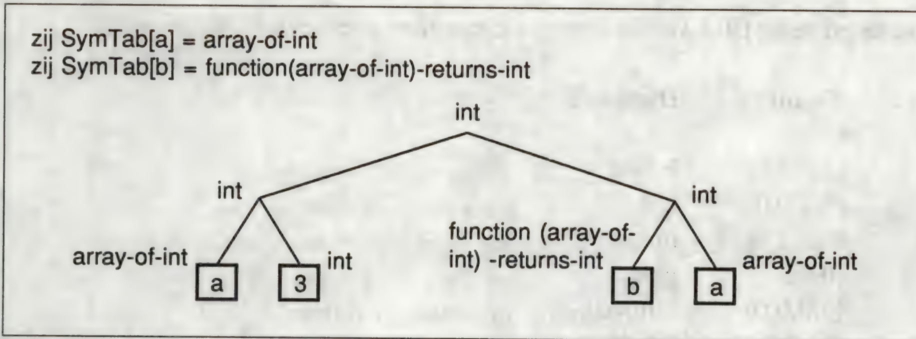
$$\text{Type[E1 (E2)]} = \begin{cases} \text{X} & \text{als Type[E1] = array-of-X} \\ & \text{en Type[E2] = int} \\ \text{Y} & \text{als Type[E1] = function(X)-returns-Y} \\ & \text{en Type[E2] = X} \\ \text{fout} & \text{in andere gevallen} \end{cases}$$

$$\text{Type[E1+E2]} = \begin{cases} \text{int} & \text{als Type[E1] = Type[E2] = int} \\ \text{fout} & \text{in andere gevallen} \end{cases}$$

De eerste regel zegt: het type van een integer constante is integer. De precieze betekenis van de tweede regel hangt af van SymTab, die we niet hebben gedefinieerd. We mogen ervan uitgaan dat die tabel rekening houdt met regels betreffende geldigheidsbereiken en correspondentie tussen argumenten en parameters. Als de identifier niet is gedeclareerd, dan moet de opdracht resulteren in een typefout of de fout 'niet gedeclareerde variabele'. In tegenstelling tot de andere regels is de tweede regel hier *context-sensitief*; dat betekent dat de regel afhangt van de omringende context, in dit geval van de declaraties. Alle andere bovenstaande regels hangen alleen af van de delen waaruit de expressie bestaat. De derde regel van de definitie geeft aan dat het type van een geïndexeerde array of een procedure-aanroep afhangt van het type van de array of het type van de procedure.

Het type van de index of de parameter moet gelijk zijn aan het overeenkomstige deel van het type van de array of de procedure. De manier waarop typecontrole wordt uitgevoerd voor expressies kan worden geïllustreerd door een parseerboom met een type bij elke knoop. Figuur 6-1 geeft een voorbeeld daarvan.

De eenvoudige definitie hierboven kan zo worden uitgebreid dat typecontrole van alle expressies en alle typen zoals die in de meeste talen voorkomen, mogelijk wordt. Zo'n definitie van de typecontrole stelt niet vast hoe de typecontrole in een compiler moet worden geïmplementeerd; de definitie wordt hier gebruikt voor een goed begrip van typecontrole. De definitiemethode die we in dit voorbeeld hebben gebruikt is ook nuttig voor het definiëren van vele andere eigenschappen van expressies en talen.



Figuur 6-1 Parseerboom en typecontrole van de expressie "a(3)+b(a)".

6.2 Voorbeelden van typecontrole

De controle van typen kan tot andere eigenschappen worden uitgebreid. Een voorbeeld is dimensie-analyse, een techniek om fouten in formules te vinden. Een ander voorbeeld is de controle op bereiken. Voor dimensie-analyse bestaat een elegante algoritme die tijdens het compileren kan worden uitgevoerd, maar het is moeilijk, zo niet onmogelijk, om het controleren van bereiken geheel tijdens het compileren te laten plaatsvinden.

Dimensie-analyse

Het onderscheid tussen een dimensie en een eenheid is belangrijk. Een dimensie is een natuurkundige grootheid die kan worden gemeten. Voor één dimensie kunnen vele eenheden bestaan; bij de dimensie lengte behoren bijvoorbeeld de eenheden cm, m, km, inch, mijl. Dimensie-analyse kan worden uitgevoerd op het niveau van dimensies of op het niveau van eenheden. De meeste mensen gebruiken de term dimensie-analyse voor beide niveaus, maar wij zullen hiertussen onderscheid maken en voor het laatste geval de term *eenhedenanalyse* gebruiken. De verzameling dimensies is een verzameling typen, met elementen als lengte, tijd, oppervlak, inhoud, snelheid en energie. Dimensies zijn zeer elegant georganiseerd in een zogenaamde *vrije Abelse groep voortgebracht door een verzameling basis-dimensies.* We zullen geen studie maken van deze algebraïsche structuur, maar een eenvoudige representatie van deze groep gebruiken. We kunnen onze ruimte van dimensies representeren als een vector van gehele getallen. Elke component van de vector staat voor één van de basisdimensies. Als voorbeeld zullen we vier basisdimensies gebruiken: lengte, tijd,

massa en geld. Elke vector van vier elementen geeft een dimensie weer.

Vector	***Dimensie***
(1,0,0,0)	lengte
(0,1,0,0)	tijd
(0,0,1,0)	massa
(0,0,0,1)	geld
(3,0,0,0)	inhoud
(1,–2,1,0)	kracht
(–3,0,1,0)	dichtheid
(0,–1,0,1)	inkomen

Er is één bepaalde dimensie, voorgesteld door (0,0,0,0), die voor grootheden zonder dimensie wordt gebruikt, zoals hoeken en verhoudingen. We hebben de volgende operaties op vectoren nodig.

(a,b,c,d) + (e,f,g,h) = (a+e, b+f, c+g, d+h)
(a,b,c,d) – (e,f,g,h) = (a–e, b–f, c–g, d–h)
n*(a,b,c,d) = (n*a, n*b, n*c, n*d)

We kunnen nu een beschrijving geven van de dimensie-analyse voor enkele eenvoudige numerieke expressies. We zullen 'Dim' gebruiken als een functie van expressies naar dimensies (voorgesteld door vectoren); we definiëren Dim inductief als volgt:

Dim[N] = (0,0,0,0)

$$\text{Dim[X+Y]} = \begin{cases} \text{Dim[X]} & \text{als Dim[X]=Dim[Y]} \\ \text{fout} & \text{in andere gevallen} \end{cases}$$

Dim[X*Y] = Dim[X] + Dim[Y]

Dim[X/Y] = Dim[X] – Dim[Y]

Dim[X**3] = 3*Dim[X]

Dim[X**Y] = waarde[Y] * Dim[X]

Merk op dat er bij machtsverheffing een probleem optreedt. Als de exponent een integer-constante is, is het gemakkelijk regels voor typecontrole te geven die tijdens het compileren kunnen worden toegepast. Maar de functie 'waarde' moet de waarde van de expressie tussen vierkante haken afleveren en dat is in het algemeen pas mogelijk tijdens het uitvoeren van het programma. Als de exponent een willekeurige expressie is, kan de dimensie van het resultaat niet tijdens de compilatie worden berekend, behalve in één speciaal geval, namelijk als de dimensie van het eerste operand (0,0,0,0) is.

Een dimensie-analyse als hier beschreven controleert wel op fouten in de dimensies, maar niet op fouten in de eenheden. Hoewel guldens en dollars dezelfde dimensie hebben, zou het fout zijn ze bij elkaar op te tellen zonder eerst de ene in de andere om te rekenen. Eenhedenanalyse wordt op dezelfde manier uitgevoerd als dimensie-analyse. Het enige verschil is dat elke component van de vector een eenheid representeert in plaats van een dimensie. De regels voor typecontrole zijn bij eenhedenanalyse hetzelfde als bij dimensie-analyse.

Voor een mooie vorm van eenhedenanalyse zou het nuttig zijn als de programmeur de verzameling eenheden kon definiëren. De compiler kan dan van die verzameling gebruik maken en een vector opstellen voor het weergeven van elke gewenste combinatie van eenheden. Het is ook mooi als de programmeur constanten met eenheden kan gebruiken. Uitgaande van die twee ideeën kunnen we op de volgende manier expliciete conversies maken van één eenheid naar een andere:

```
Eenheden zijn {voet, inch, meter}
x,y : voet
a,b : inch

a := b + x * (12 inch/voet)
```

Bij het toepassen van de regels voor typecontrole blijken de eenheden op de verwachte manier te werken. Zonder de expliciete omrekeningsfactor zou er een typefout optreden.

Het is misschien wenselijk om omrekeningsfactoren tussen verschillende eenheden op te geven en de compiler deze factoren automatisch te laten gebruiken.

Een voorbeeld van de invoer voor zo'n compiler zou kunnen zijn:

```
Eenheden zijn {km,uur,minuut,kmh}
Conversies zijn {
                  60 minuut = uur
                  1 km/uur = kmh
                  }
t1,t2 : minuut
afstand : km
snelheid : kmh

snelheid := afstand / (t1–t2)
```

De expressie aan de rechterkant van de toekenning heeft het type km/minuut. Maar de variabele 'snelheid' is van het type kmh. Om de toekenning te kunnen uitvoeren moet de compiler twee conversies inlassen: (60 minuut/uur) en (kmh uur/1 km). De computer moet op de hoogte zijn van de algebra van de eenheden en van de dimensies. Twee eenheden zijn in elkaar over te voeren dan en slechts dan als ze tot dezelfde dimensie behoren. Elke conversie definieert een betrekking tussen eenheden, namelijk dat ze tot dezelfde dimensie behoren.

Deelbereiken

Een bereik is een totaal geordende serie elementen. Een deelbereik van een bereik wordt aangegeven met behulp van een onder- en een bovengrens. Beide grenzen behoren tot het bereik, en de ondergrens is kleiner dan of gelijk aan de bovengrens. De waarden in het deelbereik zijn alle elementen tussen de onder- en de bovengrens, de beide grenzen meegerekend. Deelbereiken zijn heel nuttig gebleken voor het documenteren van programma's, doordat ze een formele manier bieden om het verwachte bereik van een waarde aan te geven. Deelbereiken zijn als gegevenstype minder algemeen aanvaard. Ada gebruikt hier het begrip *subtype*, hetgeen geen nieuw type is, maar alleen een beperking inhoudt op de waarden die mogen worden gebruikt. Het begrip subtype moet niet worden verward met de term *afgeleid type* (*derived type*) uit Ada, want dat is een nieuw type dat gelijk is aan het oude type (met naamequivalentie). Deelbereiken zijn niet als volwaardige gegevenstypen aanvaard, omdat controle tijdens de compilatie te moeilijk of te onhandig is uit te voeren.

Zelfs in talen zonder deelbereiken komen de kwesties die in deze paragraaf worden behandeld, in verkapte vorm voor. Arrays hebben meestal voor iedere dimensie een onder- en een bovengrens. Bij iedere indexeringsoperatie moet

worden gecontroleerd of de index wel tussen beide grenzen ligt. Gewoonlijk is dat een dynamische controle en wordt de fout aangegeven met 'index van array buiten de grenzen'. In sommige gevallen wordt er geen controle uitgevoerd. Een optimaliserende compiler kan vele van deze controles wegwerken door vast te stellen dat de index in het juiste bereik valt. Een compiler kan bijvoorbeeld nagaan of een constante index binnen de grenzen ligt.

Zonder verlies van algemeenheid zullen we onze aandacht beperken tot bereiken en deelbereiken van integers. Een deelbereik kan worden gerepresenteerd als een paar, bestaande uit de onder- en de bovengrens, en wel als (ondergrens, bovengrens). We zullen 'Bereik' gebruiken als een functie van expressies naar deelbereiken.

Gegeven Bereik[E]=(X,Y)
Zij Onder[E] = X
Boven[E] = Y

Bereik[N] = (N,N)
Bereik[X + Y] = (Onder[X]+Onder[Y], Boven[X]+Boven[Y])
Bereik[X – Y] = (Onder[X]–Boven[Y], Boven[X]–Onder[Y])

Voor de toekenningsopdracht is typecontrole gemakkelijk. Het deelbereik van de rechterkant moet kleiner zijn dan of gelijk aan het deelbereik van de linkerkant.

Voor X := Y
Boven[Y] $\leq$ Boven[X]
Onder[Y] $\geq$ Onder[X]

Het probleem met controle van deelbereiken wordt al duidelijk met een eenvoudige toekenning als de volgende (waarin X een variabele is in een of ander gedeclareerd deelbereik):

X := X + 1

De bovengrens van de linkerkant is kleiner dan die van de rechterkant, zodat er een fout kan optreden. Om het mogelijk te maken dat op deze fout wordt gecontroleerd tijdens de compilatie, is het nodig zulke opdrachten te verbieden of te bewijzen dat, in de context waarin de opdracht voorkomt, het bereik van X+1 een deelbereik is van dat van X. In het algemeen is deze analyse veel ingewik-

kelder dan de andere algoritmen voor typecontrole die we in dit hoofdstuk hebben gezien. De problemen zijn vergelijkbaar met die bij het bewijzen van programmacorrectheid. Een andere mogelijkheid is de controle tijdens de uitvoering van het programma te laten plaatsvinden, maar in een taal die trots is op zijn sterke typecontrole, zijn deelbereiken als gegevenstype onaanvaardbaar.

Opgaven

1. Beschouw de volgende taal:

   E ::= (zij L gelijk E in E) | EE | L
   L ::= *a* | *b* | *c* | *d* | *e*

   Geef voor deze taal een inductieve definitie van de volgende functies:

   Aantalb: levert het aantal keren dat het teken *b* in de expressie voorkomt.
   Diepte: levert het grootste aantal geneste expressies die met 'zij' beginnen.
   Waarde: levert de string die door de expressie wordt gerepresenteerd.

2. Breid het voorbeeld van typecontrole in paragraaf 6.1 uit met:

   Getypeerde pointers.
   Real waarden en constanten.
   Booleaanse waarden met '+' voor de Booleaanse operatie 'of'.
   Toekenningsopdrachten.

3. Ontwerp en implementeer de algoritmen voor eenhedenanalyse en dimensie-analyse.

4. Ontwerp en implementeer de algoritmen voor automatische conversie tussen eenheden. Ga ervan uit dat de programmeur alle eenheden moet opgeven. Laat de programmeur verder één of meer conversies aangeven. Conversies mogen redundant zijn (dat wil zeggen ze mogen af te leiden zijn van eerder gegeven conversies). Ga voor zo'n redundante conversie na of die consistent is met de verzameling eerder gegeven conversies. De algoritme moet zo uitgebreid worden, dat conversies overal worden ingevoegd waar ze van toepassing zijn.

5. Beschouw het probleem van het uitbreiden van dimensie-analyse met automatische conversies naar integers. Ontwerp zo'n systeem, maar zorg ervoor dat er geen conversies worden gegenereerd die tot deling leiden. Die moeten niet worden gebruikt om onverwacht verlies van precisie te vermijden.

6. Druk de concepten van dimensie-analyse uit met behulp van groepentheorie. Wat is het algebraïsche verband tussen eenheden en dimensies? (Vereist kennis van abstracte algebra.)

7. Wat zijn bij deelbereiken de regels voor typecontrole bij de vermenigvuldigingsoperatie?

8. Definieer een verzameling typen voor absolute en relatieve typen. Deze typen moeten bruikbaar zijn voor absolute en relatieve metingen (zoals van temperaturen) en absolute en relatieve adressen in een assembler. Wat zijn de regels voor typecontrole?

Literatuur

Inductieve definities gebaseerd op grammatica's worden op grote schaal gebruikt voor uiteenlopende doeleinden en worden soms *attribuutgrammatica's* genoemd. Voorbeelden zijn te vinden in Knuth (1968) of Stoy (1977), met toepassingen voor het definiëren van de semantiek van programmeertalen. Cleaveland (1975) en Karr en Loveman (1978) bespreken het toevoegen van technieken voor dimensie-analyse aan programmeertalen.

7

Waarden, variabelen en opslag

Het is gebruikelijk dat de begrippen referentie (dat wil zeggen variabelen en pointers) en gegevenstype in programmeertalen van elkaar gescheiden zijn. ALGOL 68 is een uitzondering, omdat daarin beide begrippen in één systeem van gegevenstypen worden gecombineerd. Maar de meeste programmeertalen brengen een scheiding aan tussen het concept gegevenstype en het begrip van variabelen en waarden. Het gebruikelijke standpunt komt in het gedrang zodra er pointers in het spel komen, waardoor er een vermenging ontstaat van het begrip gegevenstype en het begrip referentie. Het mechanisme van geheugenopslag, waaronder gedeeld gebruik, gaat dan een deel van het systeem van gegevenstypen vormen. Afgezien van deze uitzondering kunnen de begrippen gegevenstype en variabele netjes gescheiden worden. Dit hoofdstuk laat die scheiding op verschillende manieren zien. In paragraaf 7.2 wordt applicatief programmeren behandeld, waarbij blijkt dat variabelen bij het programmeren niet onmisbaar zijn. Een gegevenstype bepaalt de hoeveelheid geheugen (paragraaf 7.1), maar bepaalt niet het opslagmodel (paragraaf 7.3), het geldigheidsgebied, de opslagklasse (paragraaf 7.4) of de levensduur (paragraaf 7.4). Deze ogenschijnlijk onafhankelijke delen komen samen in een declaratie. Het hoofdstuk eindigt met een korte uitweiding over geheugensanering (garbage collection), waardoor de perceptie die de gebruiker van gegevenstypen heeft, sterk kan worden beïnvloed.

7.1 Het benodigde geheugen

Hoeveel geheugen is er nodig voor een waarde van een bepaald type? Daarop is een eenvoudig antwoord te geven. Als *n* het aantal mogelijke waarden van type

T is, dan zijn er $\log_2 n$ bits geheugen voor nodig. Als T bijvoorbeeld een type is met 32 waarden, dan zijn er vijf bits nodig om een waarde van type T te representeren. In de praktijk werkt deze analyse niet goed wegens de machine-architectuur, de woordgrootte en de in tijd inefficiënte decodering van compacte representaties. Daarbij komt dat sommige typen, zoals sequences en recursieve typen, een oneindig groot aantal waarden hebben. Een opgesomd type T met 32 elementen zal eerder in 8 of 16 bits worden opgeslagen dan in 5, omdat het opslaan en terugvinden van een waarde in een byte of geheugenwoord sneller en eenvoudiger is. Het opslaan van een gegevenstype in zo weinig mogelijk geheugen wordt *compressie* of *packing* genoemd. Daarbij moet meestal tijd tegen ruimte worden afgewogen; compressie komt neer op het loslaten van tijdsefficiëntie ten gunste van ruimte-efficiëntie.

Voor getals- en tekentypen wordt van oudsher een vast aantal bits geheugen gebruikt. Een teken beslaat in de meeste hedendaagse systemen 8 bits. Er zijn enkele computersystemen die 7 bits gebruiken, wat een voordeel is bij een woordlengte van 36 of 60 bits.* Korte integers zijn meestal 16 bits lang (inclusief één tekenbit) en lange integers zijn meestal 32 bits lang. Floating-point getallen hebben meestal een lengte tussen 16 en 36 bits. Maar wat hebben al deze aan machines ontleende maten met programmeertalen te maken? Van de meeste programmeertalen wordt graag volgehouden dat ze overdraagbaar en machine-onafhankelijk zijn. In feite zijn programmeertalen zelden onafhankelijk van de woordgrootte van een machine. Men kan niet in het algemeen zeggen dat een integer in een bepaalde programmeertaal X bits lang is. Dat hangt meestal van de machine af, niet van de taal. Gegeven de lengte (in bits) van integers kan men iets zeggen over het bereik van de integers. In sommige programmeertalen is deze afhankelijkheid van machines geformaliseerd via omgevingsinformatie.

Omgevingsinformatie is een manier om machine-afhankelijke eigenschappen vast te leggen, zoals de grootste en de kleinste integer. In ALGOL 68 bijvoorbeeld geeft een voorgedefinieerde constante als `maxint` aan wat de grootste integer waarde is. Die waarde kan in de ene machine-implementatie anders zijn dan in de andere. Hoewel het te verwachten is dat het geheugenbeslag van een gegevenstype van implementatie tot implementatie verschilt, is het verbazend dat de eigenschappen van een fundamenteel gegevenstype als integers ook van implementatie tot implementatie verschillen.

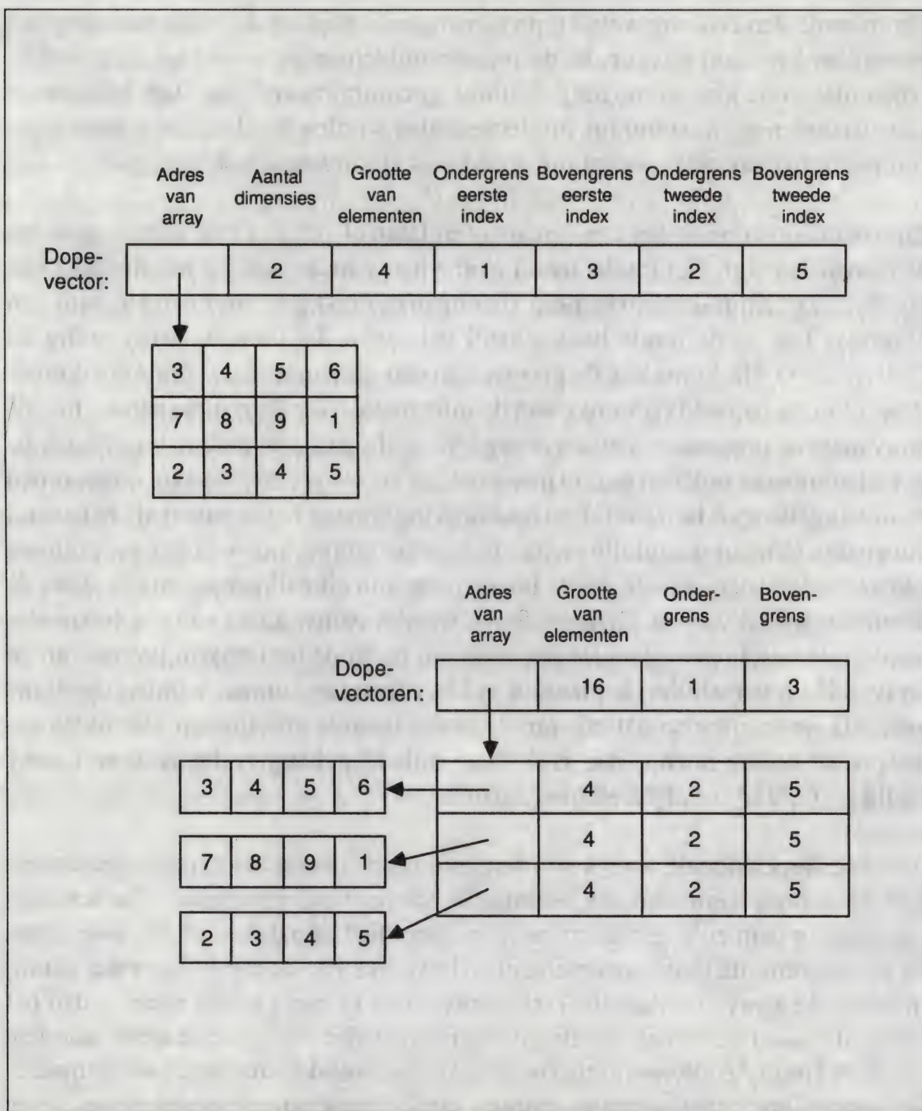
---

* Er passen 5 tekens van 7 bits in een woord van 36 bits.

Een waarde van een opgesomd type wordt gewoonlijk op dezelfde manier gerepresenteerd als een integer. In de meeste implementaties worden opgesomde typen niet in de kleinst mogelijke ruimte gecomprimeerd. Het type Boolean is een uitzondering. In sommige implementaties worden Booleaanse waarden gecomprimeerd, speciaal als het om arrays van Booleaanse waarden gaat.

Zij n het aantal dimensies van een array en laten $D_1, D_2 \ldots D_n$ de afmetingen van de dimensies zijn. Het totale aantal elementen van de array is het produkt van $D_1, D_2 \ldots D_n$. Zij E de hoeveelheid geheugen die nodig is voor één element van de array. Dan is de totale hoeveelheid geheugen die voor de array nodig is: $D_1 * D_2 * \ldots * D_n * E$. Soms kan de grootte van een dimensie niet tijdens het compileren worden bepaald en soms moet de informatie over de grootte samen met de array aan een procedure worden doorgegeven. In zulke en andere voorkomende gevallen moet er ook een *beschrijvingsvector* (*dope vector*) worden opgebouwd en in het geheugen bewaard. Een beschrijvingsvector bevat zaken als het aantal dimensies (als het aantal dimensies tijdens de uitvoering van het programma kan veranderen), de onder- en de bovengrens van elke dimensie en ten slotte de lengte (in bytes) van elk element. Soms wordt ook het adres van het eerste element in de beschrijvingsvector opgenomen, zodat de beschrijvingsvector en de array zelf op verschillende plaatsen in het geheugen kunnen worden opgeborgen. Als we aannemen dat elk van de bovenstaande grootheden vier bytes geheugen in beslag neemt, dan is de hoeveelheid geheugen die voor een array nodig is: $D_1 * D_2 * \ldots * D_n * E + 8 * n + 12$ bytes.

Als meerdimensionale arrays worden geïmplementeerd als geneste ééndimensionale arrays, verandert de benodigde hoeveelheid geheugen. De beschrijvingsvector is in zo'n geval iets eenvoudiger; die bestaat dan uit de ondergrens, de bovengrens, de lengte per element en het adres van de array. Voor een ééndimensionale array zijn dan $16 + D * E$ bytes nodig (waarin D één meer is dan het verschil tussen de boven- en de ondergrens). Voor een geneste array zijn dan $16 + D_1 * (16 + D_2 * E)$ bytes nodig, waarin D_1 het aantal elementen van de buitenste array is en D_2 het aantal elementen van de binnenste array. Merk op dat de beschrijvingsvector herhaald wordt voor elk van de binnenste arrays. Daarvoor is aanzienlijk meer ruimte nodig dan voor meerdimensionale arrays, maar er zijn ook enkele voordelen. De array is gesegmenteerd en vereist geen aaneensluitend stuk geheugen. De grenzen van de verschillende binnen-arrays kunnen van elkaar verschillen. Figuur 7-1 laat een array zien met beide soorten beschrijvingsarrays.



Figuur 7-1 Twee methoden voor het implementeren van arrays.

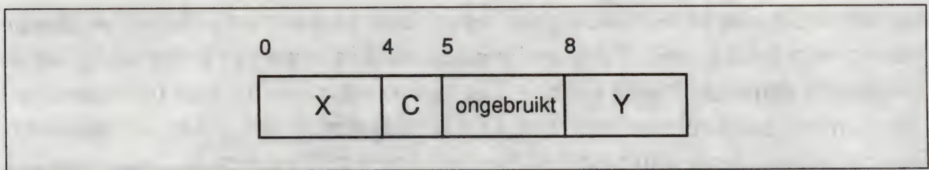
Laten we ten slotte eens kijken naar heterogene arrays. Het probleem met heterogene arrays is dat de grootte van de elementen verschillend kan zijn. Als de lengte van het grootste element bekend zou zijn, konden alle elementen in dezelfde hoeveelheid geheugen worden geplaatst en konden de oplossingen uit de vorige twee alinea's worden gebruikt. Maar die lengte is niet altijd bekend, en zelfs als die wel bekend zou zijn, zou deze methode indien de meeste ele-

menten klein zijn in verhouding tot het grootste element enorme hoeveelheden ruimte ongebruikt laten. Voor snelle indexering in een array is het nodig dat de elementen een vaste lengte hebben. Dat kan worden bereikt door het tussenvoegen van een extra referentieniveau. Dat wil zeggen: de array kan worden voorgesteld als een array van pointers. De elementen zelf kunnen dan waar dan ook in het geheugen worden opgeslagen. De beschrijvingsvector van zo'n array is hetzelfde als die van homogene arrays (op de grootte per element na), maar voor elk element van de array moet een extra pointer (meestal vier bytes) worden opgeslagen. Dat is de prijs die het kost om heterogene arrays te hebben.

Een record wordt gewoonlijk gerepresenteerd in een aaneensluitend stuk geheugen, waarin eerst het eerste veld voorkomt, en onmiddellijk daarna de verdere velden. Het minimale aantal bits dat nodig is om een record te representeren is de som van de aantallen bits die nodig zijn om elk veld van het record te representeren. Omdat de verschillende velden van het record verschillende hoeveelheden geheugen kunnen beslaan, worden velden meestal geselecteerd op basis van een afstand ten opzichte van het begin van het record. Als gevolg van de architectuur van de computer kan het zijn dat velden niet op een willekeurige bitpositie kunnen beginnen. Tekens moeten bijvoorbeeld gewoonlijk beginnen op de grens van twee bytes. Het proces van verschuiven naar de gewenste grens heet *alignering*. De noodzaak tot alignering heeft tot gevolg dat het aantal bits dat nodig is om een record te representeren soms groter is dan de som van de bits die nodig zijn om elk veld van het record te representeren. Beschouw het record

```
record
        X: REAL;
        C: CHAR;
        Y: REAL;
end record;
```

Als we ervan uitgaan dat waarden van het type REAL in vier bytes worden gerepresenteerd en dat ze moeten worden gealigneerd op een grens tussen twee viertallen bytes, dan moeten voor het representeren van bovenstaand record twaalf bytes worden gebruikt, ook al zijn er voor het opslaan van de drie velden maar negen bytes nodig. De drie extra bytes zijn nodig voor het aligneren van Y op de grens tussen twee viertallen bytes (zie figuur 7-2). Af en toe worden er trucs in programma's gebruikt, waarmee de toegang tot een veld op een ongebruikelijke manier verloopt en waarbij van een bepaalde alignering wordt uitgegaan. Zulke programma's zijn meestal niet overdraagbaar, omdat alignering een kwestie van implementatie is en afhankelijk is van de architectuur van de



Figuur 7-2 Alignering in een record (de getallen geven de afstand in bytes aan vanaf het begin van het record).

machine. Zulke programma's maken gebruik van een typelek en kunnen beter niet worden geschreven.

De hoeveelheid geheugen die voor een pointer nodig is, is onafhankelijk van datgene waarnaar de pointer wijst. De waarde van een pointer is een adres, dat op de meeste machines overeenkomt met één woord. Alleen het type pointer maakt de opslag van recursieve typen mogelijk. Beschouw het recursieve type

```
type A is record
              X: REAL;
              N: A;
          end record;
```

De hoeveelheid geheugen die voor `A` nodig is, is:

Ruimte(A) = Ruimte(X) + Ruimte(A)

De enige manier om deze recursieve vergelijking op te lossen is een oneindige hoeveelheid geheugen te gebruiken voor het representeren van `A`. Het is niet mogelijk elke mogelijke waarde van het type `A` in een eindige hoeveelheid geheugen op te slaan. Als we het type van `N` veranderen van `A` in `pointer A`, dan wordt de voor `A` benodigde hoeveelheid geheugen:

Ruimte(A) = Ruimte(X) + Ruimte(pointer)

en dat is eindig (meestal acht bytes).

De hoeveelheid geheugen die nodig is voor unions is de grootste hoeveelheid geheugen die nodig is voor één van de alternatieven plus geheugen voor de discriminant. De discriminant kan meestal gemakkelijk in één byte of één geheugenwoord worden opgeborgen.

De hoeveelheid geheugen die nodig is voor een procedure is meestal alleen de ruimte die nodig is om het begin van de procedure aan te wijzen, dus dezelfde ruimte als voor een pointer.

7.2 Werken zonder variabelen

Is programmeren zonder variabelen mogelijk? Zoals de kwestie van goto's aan het eind van de jaren 60 belangrijk was, zo is programmeren met of zonder variabelen misschien een belangrijke kwestie van de jaren 80. Laten we eens kijken wat de gevolgen zijn van programmeren zonder variabelen. Allereerst zouden er geen globale of lokale variabelen van wat voor soort dan ook zijn. Zonder variabelen is er ook geen reden om een toekenningsopdracht te hebben, omdat er geen variabelen zijn om iets aan toe te kennen. Variabelen kunnen ook in een verkapte vorm optreden. Er zouden geen invoer- en uitvoeropdrachten kunnen worden gebruikt, omdat die impliciet invoer- en uitvoerbestandsvariabelen veranderen. Procedure-aanroep by reference zou geen zin hebben, omdat veranderingen in een parameter alleen mogelijk zijn als de parameter een variabele is. Herhalingslussen zouden geen zin hebben, want het opnieuw uitvoeren van een te herhalen stuk programma is alleen wenselijk als bepaalde variabelen een andere waarde hebben en zo tot andere resultaten leiden. Er is in feite geen reden meer voor opdrachten in het algemeen. Een opdracht in een programmeertaal verandert de toestand van de machine en levert geen waarde af. Maar als er een machinetoestand is die kan worden veranderd, dan is er sprake van een verkapte variabele. Maar als er geen machinetoestand is die door een opdracht kan worden veranderd, wat is dan nog het nut van opdrachten? Dus als er geen variabelen bestaan, zijn er ook geen opdrachten.

Blijft er dan nog wel iets over om mee te programmeren? Jawel, er zijn expressies, waaronder conditionele expressies en recursieve procedures. En dat is voldoende om mee te programmeren; LISP is, in zijn oorspronkelijke vorm, zo'n programmeersysteem. In tegenstelling tot een opdracht levert een expressie een waarde op. Die waarde kan met andere waarden worden gecombineerd, aan procedures worden doorgegeven en aan operaties worden onderworpen, die weer nieuwe waarden opleveren. Er wordt wel beweerd dat programmeren zonder variabelen beter is dan met variabelen. Het is duidelijk dat zulke programma's gemakkelijker te begrijpen zijn. We hoeven niet beducht te zijn voor geheimzinnige tijdsafhankelijke veranderingen. Er zijn geen globale variabelen, geen zij-effecten en geen lussen. En die behoren tot de moeilijkste zaken voor een beginnende programmeur. Programmeren zonder variabelen wordt met

verschillende termen aangeduid, zoals *declaratief programmeren*, *applicatief programmeren* en *functioneel programmeren*. Programmeren met variabelen wordt soms *imperatief programmeren* genoemd. Een opdracht in een programmeertaal lijkt veel op een bevel. Ken 5 toe aan X. Breng 'deze string' als uitvoer over naar file Y. Deze twee zinnen zijn gebiedende of imperatieve zinnen, vandaar de term 'imperatief programmeren'.

In ALGOL 68 en Ada kunnen programma's zowel in gebiedende als in declaratieve stijl worden geschreven. Een dergelijke flexibiliteit is in deze twee talen mogelijk dank zij de aanwezigheid van constanten met een naam, die tijdens de uitvoering van het programma een waarde kunnen krijgen, van recursieve procedures en van een op expressies gerichte syntaxis. Het volgende imperatieve programma berekent de som van de getallen in een geketende lijst van getallen, en wel tot aan de eerste nul of het einde van de lijst.

```
type LIJST is pointer
        record
                WAARDE    : INTEGER;
                VOLGENDE : LIJST;
        end record;

TEL_OP_TOT_NUL: function ( A: LIJST) return INTEGER is
        SOM : INTEGER;
        PTR : LIJST;
begin
        SOM := 0;
        PTR := A;
        while PTR ≠ null and then PTR.WAARDE ≠ 0 loop
                SOM := SOM+PTR.WAARDE;
                PTR := PTR.VOLGENDE;
        end loop;
        return SOM;
end TEL_OP_TOT_NUL;
```

Om dit programma te veranderen in een programma in declaratieve stijl moeten we de variabelen en de lus omvormen tot een recursieve procedure. Die verandering kan op verschillende manieren tot stand gebracht worden, zoals op deze manier:

```
type LIJST is pointer
        record
                WAARDE    : INTEGER;
                VOLGENDE : LIJST;
        end record;
```

```
SOM_TOT_NUL: function ( A: LIJST) return INTEGER is
begin
        if A=null or else A.WAARDE=0 then
                return 0;
        else
                return A.WAARDE+SOM_TOT_NUL(A.VOLGENDE);
        end if;
end SOM_TOT_NUL;
```

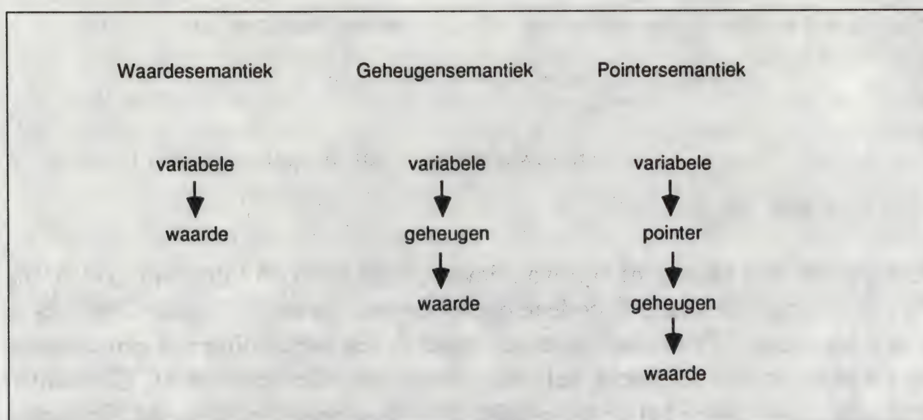
Constanten met een naam zijn waardevol omdat ze extra informatie geven die de compiler in staat stelt de code te optimaliseren. Ze maken ook het ontdekken van fouten mogelijk omdat, als er per ongeluk een toekenning aan een constante plaatsvindt, dat al tijdens het compileren kan worden ontdekt. Constanten met een naam maken het de menselijke lezer ook gemakkelijk aan de declaratie te zien dat de waarde niet zal veranderen. Maar het nut wordt kleiner als we de waarde van de constante al tijdens de compilatie moeten opgeven, zoals Pascal verlangt. De define-opdracht uit C is ook niet geschikt, omdat het daarbij gaat om substituties die tijdens het compileren worden uitgevoerd. Het is nuttig en soms noodzakelijk dat de waarde van een constante tijdens de uitvoering van het programma kan worden bepaald. Voor declaratief programmeren is het essentieel dat constanten met een naam dynamisch worden geïnitialiseerd.

7.3 Opslagmodellen

Voor applicatief programmeren is geen opslagmodel nodig, omdat daarbij het begrip toestand of geheugen niet voorkomt. Alles is op waarden gericht. Er wordt wel geheugen gebruikt bij het rekenen, maar dat is voor de gebruiker onzichtbaar. Bij imperatief programmeren worden variabelen gebruikt. Om tot een volledig begrip te komen van het gedrag en de betekenis van variabelen, pointers en operaties als toekenning, moeten we een opslagmodel construeren. We behandelen in deze paragraaf drie opslagmodellen. De namen daarvan zijn *waardesemantiek*, *geheugensemantiek* en *pointersemantiek*. Afbeelding 7-3 is een schematisch overzicht van deze drie opslagmodellen.

Waardesemantiek

Waardesemantiek is een opslagmodel dat lijkt op dat van applicatief programmeren. Een variabele wordt als een object beschouwd, niet als het adres van een geheugenplaats. Door middel van toekenningsopdrachten kunnen waarden in variabelen worden opgeslagen. Omdat het begrip geheugen hierin wordt over-



Figuur 7-3 Drie opslagmodellen.

geslagen, komen ook de begrippen gedeelde waarden, pointers en aanroep by reference niet voor. Het geheugenbeheer is voor de gebruiker onzichtbaar, net als bij applicatief programmeren. Als een taal met waardesemantiek gegevenstypen met willekeurige grootte heeft, zoals strings, dan moet het systeem voor de geheugenallocatie en de sanering van het geheugen zorgen.

Een toekenningsopdracht veroorzaakt het overdragen van een waarde naar een variabele. Om dit soort toekenning te onderscheiden van andere toekenningen noemen we deze vorm *waardetoekenning*. APL en SETL zijn voorbeelden van talen die waardesemantiek gebruiken.

Geheugensemantiek

Geheugensemantiek is het meest voorkomende opslagmodel. Talen als FORTRAN, COBOL, PL/I, C en de talen uit de familie van ALGOL en Pascal gebruiken allemaal geheugensemantiek. Het belangrijkste uitgangspunt is dat een variabele een geheugenplaats is waarin een waarde is gerepresenteerd. Een waarde heeft geen tijd- of plaatsgebonden aspecten. Maar het geheugen heeft zowel tijd- als plaatsgebonden eigenschappen. De waarde die in het geheugen is gerepresenteerd kan in de tijd veranderen. De allocatie en het vrijmaken van geheugen zijn van wezenlijk belang. Deze talen kennen soms verschillende *opslagklassen* om de allocatie en het vrijmaken van geheugen te regelen. Dit onderwerp wordt in paragraaf 7.4 verder behandeld.

Een *adres* is een middel om toegang tot een geheugenplaats te krijgen. In hogere programmeertalen worden adressen als pointers gerepresenteerd. Men zegt dat een geheugenlocatie wordt *gedeeld* als twee of meer variabelen toegang hebben tot de locatie. Deling wordt ook *aliasing* genoemd, omdat dezelfde geheugenplaats meer dan één naam heeft. Deling of aliasing kan voorkomen als een globale variabele by reference wordt doorgegeven aan een procedure. Binnen die procedure zijn de parameter en de globale variabele twee namen voor hetzelfde object. Als voor twee argumenten van een procedure dezelfde variabele wordt doorgegeven, dan delen de twee overeenkomstige parameters hetzelfde object. Een gebruikelijke aanpak van geheugendeling is het gebruik van pointers. Er kunnen willekeurig veel pointers naar één object wijzen. Aliasing is een probleem voor taalontwerpers, programmeurs en verificatie-experts. Bewijzen worden veel moeilijker als er aliasing kan optreden. Programma's zijn moeilijk te begrijpen als meerdere namen op hetzelfde object betrekking hebben. In combinatie met aanroep by reference (hier aangegeven met het sleutelwoord `var`) kan aliasing een mysterieuze werking hebben, zoals de volgende procedure laat zien:

```
MULT: procedure ( var A: array (1..10) of INTEGER;
                  var X: INTEGER) is
        J: 1..10;
begin
        for J in 1..10 loop
                A(J) := A(J)*X;
        end loop;
end MULT;

B: array (1..10) of INTEGER;

begin
        MULT(B,B(3));
        ...
```

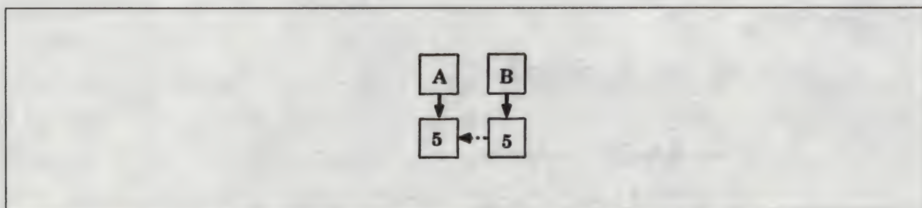
Dit is een klassiek probleem rond aliasing, dat een moeilijk te vinden fout introduceert in een ogenschijnlijk onschuldige routine voor scalaire vermenigvuldiging. Na de aanroep van de procedure worden de eerste drie elementen van `B` vermenigvuldigd met `B(3)`, maar de overige elementen worden vermenigvuldigd met het kwadraat van `B(3)`! Dit specifieke probleem met aliasing kan worden weggewerkt door het verwijderen van het attribuut `var` bij de tweede parameter.

De betekenis van een toekenningsopdracht (uitgaande van geheugensemantiek) is het kopiëren van de waarde die aan de rechterkant wordt uitgerekend in het

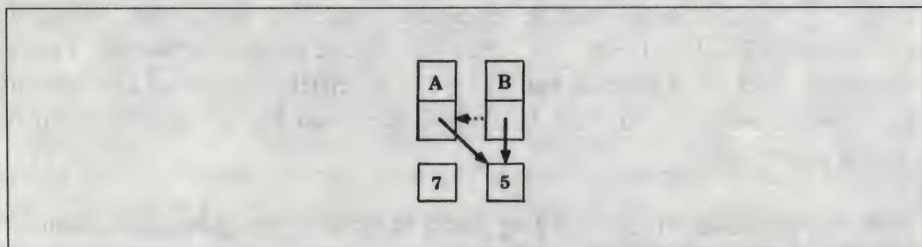
stuk geheugen waarnaar de variabele aan de linkerkant verwijst. In de meeste gevallen gedraagt deze toekenning zich hetzelfde als een waardetoekenning. De toekenning van pointerwaarden moet zorgvuldig worden bestudeerd. Bij het toekennen van een pointerwaarde wordt alleen het adres gecopieerd, niet de waarde waarnaar het adres verwijst. Deze vorm van toekenning is verwant met de gedeelde toekenning (beschreven in de volgende subparagraaf). Zolang we een pointerwaarde zuiver als een adres beschouwen, is toekenning onder geheugensemantiek gelijk aan waardetoekenning. Een voorbeeld van de verwarring die kan ontstaan is, dat een string van tekens in PL/I by value wordt toegekend, terwijl in C dan deling optreedt.

Pointersemantiek

Het laatste opslagmodel dat we hier behandelen is *pointersemantiek*. In dit model wordt een variabele beschouwd als een pointer naar een geheugenplaats die een waarde bevat. In een toekenning als A:=B wordt de pointer naar de geheugenplaats gecopieerd, niet de waarde die in de geheugenplaats is opgeslagen. Omdat A en B de geheugenplaats delen, wordt de toekenning een *gedeelde* toekenning genoemd. Onder geheugensemantiek is dit te vergelijken met de toekenning van een pointer. Het verschil is dat onder pointersemantiek het adres impliciet wordt gekopieerd (zie figuur 7-4 en 7-5).



Figuur 7-4 Waardetoekenning onder geheugensemantiek (de gestippelde pijl betekent kopiëren).



Figuur 7-5 Gedeelde toekenning onder pointersemantiek (de gestippelde pijl betekent kopiëren).

Pointersemantiek kan soms tot onverwacht gedrag leiden. Beschouw bijvoorbeeld de volgende toekenningsopdrachten, waarin A en B arrays van integers zijn:

```
A := B;
B(5) := B(5) + 1;
```

In een taal met waarde- of geheugensemantiek wordt de waarde van de array gekopieerd. Als het vijfde element van B wordt veranderd, verandert A daardoor niet.

Maar in een taal met pointersemantiek wordt de waarde van de array in de eerste toekenning niet veranderd; in plaats daarvan wordt de geheugenplaats gedeeld. Door de verhoging van de waarde van het vijfde element van B wordt ook het vijfde element van A in waarde verhoogd.

Voor eenvoudige typen als getallen kan het moeilijk zijn om onderscheid te maken tussen de verschillende opslagmodellen. Hoe kunnen we een test maken om het opslagmodel voor integers te bepalen? We kunnen iets proberen als:

```
B := 6;
A := B;
B := 4;
if A=4 then
        PRINT("POINTERSEMANTIEK");
end if;
```

Maar dit werkt niet, omdat door de derde toekenning alleen de pointer van B wordt veranderd, niet de waarde waarnaar B verwijst. Om de test effectief te maken moeten we gebruik maken van een operatie die de waarde verandert waarnaar B wijst. Maar er zijn (meestal) geen numerieke operaties die de waarde veranderen. Daardoor kunnen de basistypen in een taal met pointersemantiek hetzelfde functioneren als hun neefjes in een taal met geheugen- of waardesemantiek. We zullen om dit idee te generaliseren een gegevenstype *semantisch transparant* noemen als alle basis-operaties op het type geheugen creëren voor nieuwe waarden en geen enkele in het geheugen voorkomende waarde veranderen. We zouden dan ook kunnen spreken van *éénmaal beschreven geheugen*. Als alle gegevenstypen in een taal semantisch transparant zijn, is pointersemantiek equivalent met waardesemantiek! Vreemd genoeg worden SETL en andere talen met waardesemantiek voornamelijk geïmplementeerd met behulp van pointers en gedeelde toekenning.

In SNOBOL wordt voor alle niet-basistypen pointersemantiek gebruikt. Daaronder vallen arrays, tabellen en gegevensstructuren. In sommige talen wordt een combinatie van opslagmodellen gebruikt. In de taal C wordt een array als een pointer opgevat. Als gevolg daarvan heeft C pointersemantiek. Maar voor andere gegevensstructuren, zoals records, wordt geheugensemantiek gebruikt. Aankomende programmeurs kunnen in SNOBOL en in C tot de ontdekking komen dat sommige array-operaties verrassende resultaten leveren. Een ander verschil tussen COBOL en C ligt in het opslagmodel dat voor strings van tekens wordt gebruikt. In C wordt een string beschouwd als een array van tekens en daarom is op strings in C pointersemantiek van toepassing. In SNOBOL is een string geen array. Strings zijn in SNOBOL semantisch transparant en vallen dus onder waardesemantiek.

7.4 Opslagklassen, levensduur van variabelen en geheugensanering

Een waarde heeft geen levensduur, maar bestaat onafhankelijk van tijd of ruimte. Het getal 4 is niet gecreëerd en wordt niet vernietigd, en heeft ook geen verblijfplaats ergens in de ruimte. Een variabele in een programmeertaal komt daarentegen op en verdwijnt weer, en heeft (onder geheugensemantiek) een bepaalde locatie in het geheugen, waar waarden kunnen worden weergegeven. De *levensduur* van een variabele is de tijdsruimte tussen het moment waarop er ruimte voor de variabele wordt gealloceerd en het tijdstip waarop die ruimte weer wordt vrijgegeven. De levensduur van een variabele wordt bepaald door de *opslagklasse*. Gedurende de levensduur van een variabele is de toegang ertoe geoorloofd, maar daarvóór of erna levert dat een fout op. De levensduur van een variabele hangt vaak samen met het geldigheidsgebied (de *scope*). In traditionele talen met een blokstructuur begint de levensduur van een variabele als het blok wordt binnengegaan waarin de variabele wordt gedeclareerd; de levensduur eindigt dan bij het verlaten van dat blok. Deze opslagklasse wordt in C en PL/I *automatic* genoemd en in ALGOL 68 *loc*. Elke keer dat het blok wordt binnengegaan wordt er geheugen gealloceerd voor een nieuwe, verse variabele. Als het blok meermalen wordt binnengegaan worden er ook meerdere exemplaren van de variabele gecreëerd. Deze conventie maakt het mogelijk recursieve procedures op een natuurlijke manier uit te drukken. Maar het is daarbij niet gemakkelijk om functies te schrijven die rekening houden met het verleden, dat wil zeggen een functie die waarden wil gebruiken die bij voorafgaande aanroepen van de functie zijn berekend. Zo'n functie moet de beschikking hebben over geheugen om deze waarden van de ene aanroep tot de vol-

gende te bewaren. Generators van pseudo-toevalsgetallen moeten bijvoorbeeld vaak enige informatie over de vorige aanroep bewaren. Om dat mogelijk te maken hebben we een globale variabele nodig, die ook vanuit andere plaatsen toegankelijk is, of een locale variabele met een levensduur die uitgaat boven die van het blok. Zulke locale variabelen worden soms *statisch* genoemd. In ALGOL 60 werden ze *own* variabelen genoemd. De levensduur van zo'n variabele is gelijk aan die van het gehele programma. Het geheugen ervoor wordt gealloceerd als het programma begint en wordt pas vrijgegeven als het programma klaar is. In FORTRAN is dit het enige soort geheugen dat beschikbaar is. In FORTRAN wordt het geheugen voor alle variabelen gealloceerd bij het begin van het programma en vrijgegeven aan het eind. Om die reden zijn recursieve procedures in FORTRAN niet toegestaan.

De laatste veel voorkomende opslagklasse is dynamisch geheugen. Dat geheugen wordt in PL/I *based* geheugen genoemd, in ALGOL 68 *heap* geheugen. Daarbij wordt er voor een variabele geheugen gealloceerd door een expliciete opdracht of procedure-aanroep. De programmeur draagt de verantwoordelijkheid voor het geheugenbeheer voor zulke variabelen en moet zelf geheugen alloceren en vrijgeven op het ogenblik dat dat nodig is. Dynamisch geheugen is nuttig voor het opbouwen van lijststructuren en arrays van wisselende omvang.

Onlangs heeft G.V. Cormack voorgesteld de programmeur meer zeggenschap te geven over de levensduur en het geldigheidsgebied van variabelen. Hij noemt andere vormen van levensduur, zoals een *bevattende* (*containing*) levensduur, die gelijk is aan die van het blok dat het blok van de declaratie omvat. De programmeur zou in feite in staat zijn elke gewenste levensduur te specificeren voor elke variabele, ook voor expliciet gealloceerde variabelen.

7.5 Geheugenbeheer

We hebben het gehad over de hoeveelheid geheugen die nodig is voor het weergeven van waarden, over wanneer geheugen wordt gealloceerd en vrijgegeven en over opslagmodellen, maar we hebben nog niet gezien hoe het geheugen wordt beheerd. Deze paragraaf bevat een korte introductie tot enkele technieken van geheugenbeheer die in programmeertalen veel voorkomen. Statisch geheugen is het gemakkelijkst te beheren, omdat het maar één keer wordt gealloceerd en tot het einde toe in stand blijft. Een dergelijke eenvoud van geheugenbeheer is in FORTRAN te vinden, omdat alle geheugen in FORTRAN statisch is.

Het op één na gemakkelijkst is het automatisch geheugen. Dat wordt gewoonlijk beheerd met behulp van het stack-model voor talen met blokstructuur. Automatisch geheugen wordt gealloceerd en vrijgegeven met behulp van een run-time stack. Als een blok wordt binnengegaan, wordt het geheugen dat voor dat blok nodig is, op de run-time stack geplaatst. Bij het verlaten van het blok wordt het geheugen weer van de stack gehaald. Als we afzien van hangende referenties, zijn variabelen niet meer toegankelijk nadat ze van de stack zijn gehaald, zodat ze kunnen worden vrijgegeven op het moment dat het blok wordt verlaten. Omdat alleen het actieve blok dat het laatst is binnengegaan, vrijgegeven kan worden, werkt het stackmodel goed.

Het moeilijkst te beheren soort geheugen is het dynamisch geheugen. Omdat er van te voren geen volgorde vaststaat waarin geheugen wordt gealloceerd en vrijgegeven, kan het geheugen niet worden beheerd met een stack of een andere eenvoudige gegevensstructuur. Het probleem is zo moeilijk dat een aantal talen de verantwoordelijkheid niet op zich nemen en aan programmeurs de zorg overlaten om hun eigen geheugen in orde te houden. In Pascal, PL/I en C wordt dynamisch geheugen expliciet vrijgegeven met opdrachten die de programmeur in het programma moet zetten. Er moet voor worden gezorgd dat er geen hangende referenties worden gebruikt. Als alles wat wordt gealloceerd, in gebruik blijft tot aan het eind van het programma, behoeft er geen geheugen te worden vrijgemaakt. Maar in veel programma's is het nodig eenzelfde stuk geheugen meermalen te gebruiken. In zulke programma's moet de programmeur ervoor zorgen dat hij geheugen vrijgeeft als het niet meer nodig is.

In andere talen, zoals ALGOL 68, SNOBOL en Ada, is er moeite gedaan om ongebruikt dynamisch geheugen automatisch op te ruimen. Dat proces wordt meestal *geheugensanering* of *garbage collection* genoemd. Het grootste probleem bij geheugensanering is het bepalen van de objecten die niet langer door het programma worden gebruikt. Als geen enkele variabele in het programma meer toegang heeft tot een bepaald object (zelfs niet via een aantal niveaus van indirectie), dan kan het object nooit meer worden gebruikt en kan het voor nieuw gebruik worden vrijgegeven. Ontoegankelijke objecten kunnen onder andere worden gevonden met een *markeringsalgoritme*, die uit de volgende stappen bestaat:

1. Markeer alle objecten in het systeem. Voor het markeren is één bit geheugen nodig voor elk object in het systeem.
2. Volg voor iedere pointer-variabele in het programma alle paden (dat wil zeggen volg elke pointer door alle geketende gegevensstructuren heen).

Verwijder bij elk object op het pad de markering.

3. Alle gemarkeerde objecten zijn ontoegankelijk vanuit variabelen in het programma en kunnen daarom worden vrijgegeven voor nieuw gebruik.

Deze methode verlangt een aantal dingen van de taal. De methode veronderstelt dat alle gecreëerde objecten gevonden kunnen worden en dat het systeem weet waar alle pointers in elk object voorkomen. Deze twee veronderstellingen kunnen in een sterk getypeerde taal best worden gemaakt. Maar in een taal met typelekken is het mogelijk pointers te verbergen en daarmee de hierboven beschreven methode onbruikbaar te maken. Deze methode heeft het voordeel dat de extra kosten alleen bestaan uit de tijd voor de sanering zelf en uit enig extra geheugen voor het markeren van objecten en voor het bepalen van de plaats van pointers in gegevensstructuren.

In een tweede methode, die werkt met een *referentieteller* (*reference count*), wordt aan elk object een nieuw veld toegevoegd om bij te houden hoeveel pointers ernaar wijzen. De referentieteller wordt bijgehouden door steeds 1 bij te tellen of af te trekken bij elke toekenning of kopiëring van een pointer. Bij de pointertoekenning

```
P := Q;
```

wordt de referentieteller van het object waarnaar `P` vóór de toekenning wees, met 1 verlaagd en wordt de referentieteller van het object waarnaar `Q` wijst met 1 verhoogd. De methode met de referentieteller is een veel snellere vorm van geheugensanering dan de markeringsalgoritmen, omdat het voor het verzamelen van al het ongebruikte geheugen voldoende is één keer door het geheugen te gaan. Elk object met een referentieteller gelijk aan nul kan worden vrijgegeven voor nieuw gebruik. Een probleem van de methode is dat cycli niet altijd worden opgeruimd. In onderstaand programma wordt het object waarnaar `Q` wijst nooit opgeruimd, omdat de referentieteller nooit nul wordt, ook al maakt de laatste toekenningsopdracht het object ontoegankelijk.

```
type LIJST is pointer
        record
                WAARDE    : INTEGER;
                VOLGENDE : LIJST;
        end record;

Q: LIJST;
```

```
begin
        Q            := new LIJSTKNOOP;
        Q.VOLGENDE := Q;
        Q            := null;
end
```

Een ander belangrijk probleem bij het beheren van dynamisch geheugen is fragmentatie. *Fragmentatie* treedt op als al het vrije geheugen in het systeem voorkomt in de vorm van kleine stukjes die geen van alle groot genoeg zijn voor een enkele allocatie. Die situatie kan optreden als er veel geheugenblokken van wisselende lengte worden gealloceerd en vrijgegeven. Eén van de methoden van geheugensanering, namelijk *compressie* (*compactificatie*), doet daar iets aan, door alle vrije stukken geheugen aaneen te voegen. Daarbij moeten dan natuurlijk alle pointers worden bijgewerkt die naar blokken wijzen die in het geheugen worden verschoven.

In traditionele, op stacks gebaseerde talen vindt het merendeel van het alloceren en vrijgeven op een stack plaats. Die aanpak voorkomt de problemen van dynamisch geheugenbeheer, maar heeft weer het probleem van hangende pointers (of procedures, zie paragraaf 4.1). *Retentie* is een manier van geheugenbeheer die een alternatief biedt voor de strategie in talen die op stacks gebaseerd zijn. Geheugenbeheer dat uitgaat van retentie geeft bij het verlaten van een blok niet alle geheugen vrij, maar bewaart geheugen dat later misschien nog wordt gebruikt. Daarvoor zijn aanzienlijke veranderingen nodig in het eenvoudige model van geheugenbeheer met stacks. Voor talen met retentie is dynamisch geheugenbeheer nodig.

Opgaven

1. We hebben verschillende manieren gezien om beschrijvingsvectoren en arrays op te bergen. Vergelijk de hoeveelheden geheugen die nodig zijn, uitgaande van de volgende situaties:

   Het aantal dimensies ligt vast, de grootte van een dimensie is variabel.
   Het aantal en de grootte van de dimensies is variabel, de grootte per element is constant.
   Het aantal dimensies ligt vast, de grootte van een dimensie en van een element is variabel.

2. Herschrijf de volgende procedures zonder variabelen.

```
type LIJST is access
    record
        WAARDE:   INTEGER;
        VOLGENDE: LIJST;
    end record;

MINIMUM: function ( A: LIJST ) return INTEGER is
    KLEINSTE: INTEGER := 0;
    PTR: LIJST;
begin
    if A ≠ null then
        KLEINSTE := A.WAARDE;
        PTR := A.VOLGENDE;
        while PTR ≠ null loop
            if KLEINSTE > PTR.WAARDE then
                KLEINSTE := PTR.WAARDE;
            end if;
            PTR := PTR.VOLGENDE;
        end loop;
    end if;
    return KLEINSTE;
end MINIMUM;

GEMIDDELDE: function ( VAN,TOT: INTEGER;
                       A: array () of REAL ) is
    SOM: REAL := 0.0;
    J: INTEGER;
begin
     for J in VAN..TOT loop
       SOM := SOM + A(J);
    end loop;
    if VAN>TOT then
        REPORT_ERROR("GEMIDDELDE ZONDER BEREIK AANGEROEPEN");
        return 0;
    else
        return SOM/(1+TOT-VAN);
    end if;
end GEMIDDELDE;

GGD: function (X,Y: INTEGER) return INTEGER is
    R: INTEGER;
begin
    loop
        R := Y;
        while R>=X loop
            R:=R-X;
        end loop;
        if R=0 then
            return X;
        end if;
```

```
        Y := X;
        X := R;
    end loop;
end GGD;
```

3. Schrijf een Pascal-programma om te laten zien dat Pascal geen pointer-semantiek gebruikt. Is het mogelijk het verschil tussen geheugen- en waardesemantiek vast te stellen? Zo ja, schrijf dan een Pascal-programma om het verschil te laten zien. (Aanwijzing: gebruik parameters die by reference worden doorgegeven.)

4. Schrijf een SNOBOL-programma dat aantoont dat in SNOBOL voor niet-basistypen pointer-semantiek wordt gebruikt. Bedenk een manier om te laten zien dat voor basistypen al of niet pointer-semantiek wordt gebruikt.

5. Wanneer zou een opslagklasse `containing` nuttig zijn (zie paragraaf 7.4)?

6. Geef een techniek aan om recursieve procedures te schrijven, uitgaande van uitsluitend statisch geheugen.

7. Zij `set of x` een typeconstructor voor het declareren van verzamelingen van het type `x`. Hoeveel geheugen is er voor dit type nodig?

8. Geef aan hoe een type `type` in een taal kan worden gerepresenteerd. Hoeveel geheugen is ervoor nodig? Maak een vergelijking tussen talen met een eindig aantal typen en met oneindig veel typen.

Literatuur

MacLennan (1982) geeft een duidelijke beschrijving van de verschillen tussen waarden en objecten. Henderson (1980) is een goede inleiding tot applicatief programmeren. Er bestaan ook jaarlijkse conferenties over LISP en applicatief programmeren, waarin ook nieuwe machine-architecturen worden besproken. LISP en vele andere applicatieve talen vinden hun oorsprong in de lambda-calculus, ontwikkeld door Church in samenwerking met anderen. Functioneel programmeren in de geest van FP, de taal van Backus (1978b), is gebaseerd op het idee van combinatoren, een begrip dat voor het eerst is voorgesteld door Schnfinkel en onafhankelijk door Curry (zie Curry en Feys, 1958). Zowel in functioneel programmeren als in de combinatorische logica van Curry worden functies gedefinieerd door het combineren van andere functies, in plaats van

met behulp van parameters. Cormack (1983) komt met nieuwe regels voor geldigheidsgebied en levensduur. Algoritmen voor geheugensanering en compressie worden beschreven in Knuth (1973), Standish (1980), Cohen (1981) en Cohen en Nicolau (1983). Berry et al. (1978) bespreken retentie en wat daarmee samenhangt.

Deel III

Gegevens-abstracties

8

Abstracte gegevenstypen

Een van de belangrijkste programmeerconcepten die in de jaren zeventig zijn geïntroduceerd, is het abstracte gegevenstype. Abstracte gegevenstypen geven ons een nieuwe manier in handen voor de opzet en het ontwerp van programma's die zowel betrouwbaarder als gemakkelijker te veranderen zijn. Hoewel de termen *abstracte gegevenstypen* en *gegevensabstractie* gebruikelijk zijn, zullen we hier de term *gegevenstype* gebruiken, omdat er meestal geen onderscheid is. Op dezelfde manier wordt het gebruik van procedures soms procedurele abstractie genoemd. Al in een vrij vroeg stadium werd ontdekt dat procedures onontbeerlijk zijn voor goed programmatuurontwerp en tegenwoordig is procedurele abstractie het fundament voor kwaliteitsprogrammatuur die modulair en meervoudig bruikbaar is.

Gegevensabstracties vormen een uitbreiding van dit idee. Soms denken mensen ten onrechte dat een gegevensabstractie niet meer is dan een verzameling samenhangende procedures. Anderen denken dat een gegevensabstractie een constructie is uit een academische programmeertaal. De volgende twee hoofdstukken beschrijven de motivatie achter gegevensabstracties en laten zien hoe ze in programmatuur worden gebruikt. In paragraaf 8.1 wordt zowel het algemene idee als de constructie in programmeertalen bestudeerd. In paragraaf 8.2 wordt de vroege evolutie van abstracte gegevenstypen beschreven. Paragraaf 8.3 en 8.4 laten enkele manieren zien om abstracte gegevenstypen te implementeren in talen met en zonder faciliteiten voor abstracte gegevenstypen. In het volgende hoofdstuk worden grotere voorbeelden van gegevensabstracties gepresenteerd.

8.1 Concept en constructie

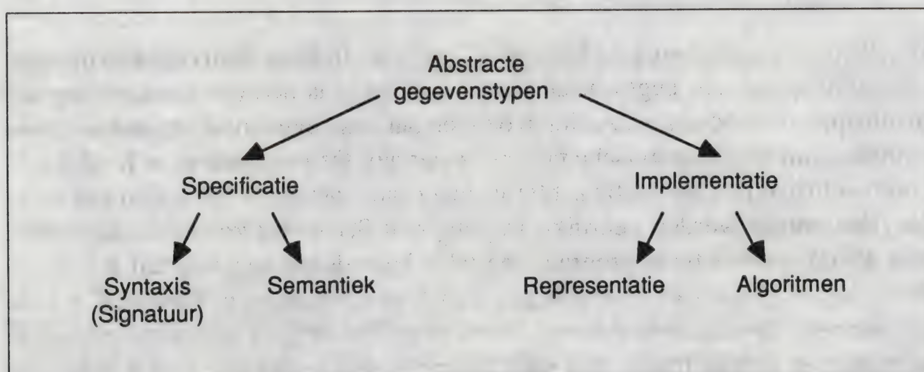
Het wezenlijke idee achter abstracte gegevenstypen is:

> Het scheiden van
> het gebruik van een gegevenstype
> en de implementatie ervan.

Het idee om een dergelijke scheiding aan te brengen is niet nieuw, maar het is wel nieuw om die scheiding te betrekken op gegevenstypen en om de scheiding te formaliseren en in programmeertalen op te nemen.

Een abstract gegevenstype kan in vier hoofdonderdelen worden gesplitst, zoals figuur 8-1 laat zien. De eerste twee onderdelen, de syntaxis en de semantiek, definiëren hoe een applicatieprogramma het abstracte gegevenstype gebruikt. De andere twee onderdelen, representatie en algoritmen, definiëren een mogelijke implementatie van het abstracte gegevenstype. Om het gegevenstype te gebruiken is het niet nodig de implementatie te kennen. Het is zelfs belangrijk van kennis van de implementatie geen gebruik te maken. Dit punt is de spil waarom gegevensabstractie draait.

De syntaxis van het abstracte gegevenstype specificeert alle operatorsymbolen of functienamen, het aantal en de typen van de operanden en het type van de waarde die wordt afgeleverd. De syntaxis geeft niet aan welke waarde bij een bepaalde invoer als resultaat wordt gegeven. De syntaxis wordt gegeven in de declaratie van een procedure. Neem als voorbeeld het gegevenstype *bag van integers*, met drie operaties: `INSERT`, `REMOVE` en `IN`. We kunnen de syntaxis ge-



Figuur 8-1 Abstracte gegevenstypen.

makkelijk specificeren door voor elke operatie de kopregel van de procedure te geven.

```
INSERT:   function (X: INTEGER; B: BAG) return BAG;
REMOVE:   function (X: INTEGER; B: BAG) return BAG;
IN:       function (X: INTEGER; B: BAG) return BOOLEAN;
```

De syntactische specificatie van een type beschrijft niet het gedrag van het type; tot nu toe is de semantiek van de operaties INSERT, REMOVE en IN niet gegeven (hoewel de namen van de operaties een bepaald gedrag kunnen doen vermoeden). De semantiek van een gegevenstype specificeert voor elke operatie van het gegevenstype welke waarde als resultaat wordt afgeleverd bij elke mogelijke invoer. De semantiek van het type bag kan in een natuurlijke taal of in een formele taal worden beschreven, en wel in twee verschillende stijlen.

De eerste stijl specificeert een representatie op hoog niveau en specificeert vervolgens operationele definities op de representatie. De stijl van de definitie moet niet worden verward met de implementatie. De definitie van de semantiek van het type bag zou er in de eerste stijl in het Nederlands beschreven zo kunnen uitzien:

Een bag is een zak waar een aantal objecten in kan worden gestopt of uit kan worden gehaald. Zo'n zak kan worden beschouwd als een sequence van getallen.

1. Een bag is aanvankelijk leeg en wordt dan als de lege sequence gerepresenteerd.

2. Een INSERT operatie plaatst een nieuw getal aan het eind van de sequence.

3. De bag kent een maximaal aantal toegestane objecten. Getallen die aan een volle bag worden toegevoegd, worden genegeerd.

4. Een REMOVE operatie zoekt de sequence af naar een voorkomen van het betreffende getal. Wordt dat gevonden, dan wordt het uit de sequence verwijderd. Als het meer dan eens voorkomt, wordt slechts één voorkomen verwijderd, de andere blijven bewaard. Als het getal niet voorkomt, wordt de bag niet veranderd.

5. De operatie IN levert true als resultaat als het getal ergens in de sequence voorkomt; anders wordt false als resultaat afgeleverd.

In de tweede stijl wordt de relatie tussen de verschillende operaties beschreven. Dat gebeurt meestal door het geven van axioma's of uitspraken over het gedrag van de operaties.

In het Nederlands kan de definitie in de tweede stijl zo verlopen:

Beschouw een bag waarop een aantal INSERT en REMOVE operaties hebben plaatsgehad. In veel gevallen doet de volgorde van de operaties niet ter zake.

1. Twee INSERT operaties of twee REMOVE operaties kunnen verwisseld worden.

2. Een INSERT en een REMOVE operatie kunnen worden verwisseld als ze betrekking hebben op verschillende getallen.

3. Als een INSERT operatie een getal toevoegt dat weer wordt weggenomen door de erop volgende REMOVE operatie, dan kunnen die twee operaties worden geschrapt.

4. Een REMOVE operatie uitgevoerd op een lege bag kan worden geschrapt.

5. Als er na het schrappen volgens bovenstaande regels alleen INSERT operaties overblijven en er meer INSERT operaties overblijven dan de maximale omvang van de bag, dan kunnen de INSERT operaties die aan het eind te veel zijn, worden geschrapt.

6. De operatie IN kan gemakkelijk worden gedefinieerd voor bags waarbij geen operaties meer kunnen worden geschrapt. De operatie IN levert true als resultaat als er een INSERT operatie overblijft voor dat getal; anders is het resultaat false.

Er bestaat een grote verscheidenheid aan formele beschrijvingsmethoden voor het specificeren van de syntaxis en van de semantiek van abstracte gegevenstypen. Enkele van die methoden worden in hoofdstuk 10 besproken.

De implementatie van een abstract gegevenstype bestaat uit de representatie en de algoritmen. De representatie geeft aan hoe waarden van het abstracte type in het geheugen moeten worden gerepresenteerd. Zo kan het type bag worden gerepresenteerd als een array van integers of als een geketende lijst van integers. De algoritmen geven aan hoe de operaties worden geïmplementeerd. Ze specificeren hoe de representatie precies moet worden gebruikt en gemanipuleerd. Voor het verkrijgen van een werkende implementatie moeten zowel de representatie als de algoritmen in een of andere programmeertaal worden gecodeerd. Voor een implementatie is het niet nodig dat de syntaxis en de semantiek geformaliseerd worden.

Het scheiden van het gebruik van een gegevenstype (gespecificeerd door de syntaxis en de semantiek) en de implementatie is om een aantal redenen belangrijk, onder andere met het oog op correctheid. Soms is het van het grootste be-

lang om van een bepaald deel van een programma de correctheid te bewijzen. In sommige gevallen heeft de noodzaak tot correctheid betrekking op het implementeren en manipuleren van bepaalde gegevens of een bepaald gegevenstype. Abstracte gegevenstypen vereenvoudigen zulke correctheidsproblemen, doordat de code die echt aan de gegevens komt, op een bepaalde plaats komt te staan, terwijl de gegevens onbereikbaar worden gemaakt voor de rest van het applicatieprogramma. De rest van het programma mag de gegevens alleen gebruiken via de welgedefinieerde operaties op de gegevens. Deze beperking wordt soms *protectie* of *inkapseling* genoemd. De gegevens die door de abstractie worden verborgen, worden soms *privé* of *beschermde* gegevens genoemd. Het afdwingen van zulke beperkingen is een belangrijke eigenschap van de faciliteiten voor abstracte gegevenstypen die in sommige programmeertalen aanwezig zijn. Zonder zulke faciliteiten kan het afdwingen van de beperkingen niet worden geautomatiseerd, waardoor de regels gemakkelijker kunnen worden overtreden.

Hoe zou het overtreden van zo'n beveiligende maatregel eruitzien? Laten we uitgaan van het type bag, gerepresenteerd als een array. Neem aan dat iemand om een of andere reden het eerste element van die array zou willen kennen, en dat de array niet beschermd zou zijn. De programmeur zou de array dan gewoon met de constante 1 kunnen indiceren en zo de waarde krijgen en kunnen afdrukken. Welke informatie wordt er dan afgedrukt? Het zou het eerste getal kunnen zijn dat ooit aan de bag is toegevoegd. Het kan ook iets anders zijn, afhankelijk van de implementatie. Als de implementatie wordt veranderd, wordt er misschien iets anders afgedrukt. In dit voorbeeld krijgt de programmeur gewoon geen relevante informatie. De overtreding van de regels lijkt misschien onschuldig, maar door veranderingen in de implementatie zou het ogenschijnlijk goed lopende programma kunnen gaan haperen. Een nog kwalijker situatie treedt op als de programmeur de waarde van het eerste element van de array zou veranderen. In dat geval wordt de bag bedorven en zal de juiste waarde van de bag niet meer foutloos gerepresenteerd zijn. Het aantonen van de correctheid van zo'n programma is veel ingewikkelder, doordat de correctheid van het gegevenstype niet beperkt blijft tot de operaties. Elk deel van het applicatieprogramma dat via ongeoorloofde operaties aan de gegevens komt, moet ook bij het correctheidsbewijs worden betrokken.

Een abstract gegevenstype kan op meer dan één manier geïmplementeerd zijn. Figuur 8-1 is dan ook niet helemaal accuraat, omdat er maar één implementatie in voorkomt. Een abstract gegevenstype kan op elk gewenst aantal manieren worden geïmplementeerd, en elke mogelijke implementatie kan worden ge-

bruikt. Welke implementatie moet dan in een concrete situatie gebruikt worden? Als we ervan uitgaan dat alle implementaties correct zijn, ligt het enige verschil tussen de implementaties in de efficiëntie. Een bepaalde implementatie kan beter zijn dan de andere omdat die minder plaats of tijd nodig heeft. De efficiëntie van gegevenstypen kan van applicatie tot applicatie verschillen. De efficiëntie kan ook veranderen op grond van verschillen in de invoer voor één enkel programma. Daarom zal de keus van een implementatie voor elke situatie verschillend zijn. Er is misschien geen implementatie die in alle situaties het beste is. Eén van de belangrijkste redenen om abstracte gegevenstypen te gebruiken is de mogelijkheid van implementatie te wisselen. Als er een efficiëntere implementatie wordt gevonden, kan die gemakkelijk de plaats van de oude implementatie innemen.

Om het verschil te demonstreren tussen de traditionele manier van problemen oplossen en de aanpak met abstracte gegevenstypen, zullen we eens kijken naar een programma dat *Rubiks kubus* oplost. Dat is een kubus die bestaat uit 3×3×3 kleinere kubusjes. Elke schijf van negen kubusjes kan onafhankelijk van de twee ermee evenwijdige schijven gedraaid worden. Dat geldt voor elke schijf in elk van de drie dimensies. In de uitgangsstand bestaat elk buitenvlak van de kubus uit negen vierkantjes van dezelfde kleur. Elk buitenvlak van de kubus is verschillend gekleurd. Als de schijven worden gedraaid, krijgen de buitenvlakken een gemengde kleurenopbouw. Als een paar van de schijven zomaar een paar keer zijn gedraaid, is niet duidelijk meer te zien welke schijven we moeten draaien om weer in de uitgangsstand terug te komen. Veel mensen in de hele wereld hebben aan deze puzzel plezier beleefd. Het is ook een goed hulpmiddel bij het onderricht in groepentheorie.

Laten we eens proberen een programma te schrijven dat, gegeven een kubus in een of andere willekeurige stand, een serie draaiingen genereert die de kubus in de uitgangsstand terugbrengt. Om dit probleem op te lossen moeten we eerst een manier vinden om de kubus te representeren. Er zijn verschillende methoden mogelijk, maar het ligt het meest voor de hand om de kubus te representeren als een driedimensionale array, geïndiceerd met vlak, rij en kolom. In elk element van de array wordt één van de zes kleuren gerepresenteerd, en wel die van het vierkantje dat door de indices wordt aangewezen. De declaratie van de kubus zou dan kunnen zijn:

```
RUBIK: array (1..6, 1..3, 1..3) of 1..6;
```

Deze declaratie beschrijft de kubus niet erg duidelijk. De declaratie is onleesbaar omdat de getallen de lezer niet veel zeggen.

Maar in dat gebrek kunnen we gemakkelijk voorzien door het gebruik van opgesomde typen:

```
type KLEUR   is (ROOD, BLAUW, GROEN, GEEL, ZWART, ORANJE);
type VLAK    is (PLAFOND, BODEM, LINKS, RECHTS, VOOR, ACHTER);
type RIJ     is (BOVEN, MIDDEN, ONDER);
type KOL     is (LINKS, MIDDEL, RECHTS);
type KUBUS   is array(VLAK, RIJ, KOL) of KLEUR;

RUBIK: KUBUS;
```

Een dergelijke declaratie beschrijft de betekenis van de typen op een prettige manier. Dit is de standaardmanier om een programma voor het oplossen van de kubuspuzzel aan te pakken. Hiervan uitgaand kan de programmeur dan een aantal operaties definiëren of direct beginnen met het oplossen van de puzzel.

Met abstracte gegevenstypen is de aanpak anders. De vragen worden in een andere volgorde behandeld. De eerste vraag die in de standaardaanpak wordt gesteld heeft betrekking op het object zelf, niet op de manier waarop het wordt gebruikt. Bij een aanpak met abstracte gegevenstypen is de eerste vraag: wat zijn de operaties, welke operaties zijn er nodig voor het oplossen van de kubus? Sommige operaties geven ons informatie, andere brengen veranderingen aan in de kubus. Hier is een mogelijke verzameling operaties:

```
type KLEUR     is (ROOD, BLAUW, GROEN, GEEL, ZWART, ORANJE);
type VLAK      is (PLAFOND, BODEM, LINKS, RECHTS, VOOR, ACHTER);
type RIJ       is (BOVEN, MIDDEN, ONDER);
type KOL       is (LINKS, MIDDEL, RECHTS);
type KUBUS     is private;
type RICHTING  is (KLOK, TEGEN_KLOK);

KLEUR_VAN:     function  (V:VLAK; R:RIJ; K:KLEUR; T:KUBUS)
                    return KLEUR;
DRAAI:         procedure (V:VLAK; R: RICHTING; in out T:KUBUS);
INITIALISEER: function  return KUBUS;
```

De gegevenstypen `KLEUR`, `VLAK`, `RIJ` en `KOL` blijven gelijk. Het gegevenstype `KUBUS` is `private`, hetgeen betekent dat het niet toegankelijk is van buiten de gegevensabstractie. Er komt een nieuw gegevenstype `RICHTING` bij om de richting van een draaiing op te geven. Dit type is hier nodig, omdat de operaties op de kubus geformaliseerd worden en één van de parameters voor het manipuleren van de kubus deze informatie nodig heeft. Vervolgens wordt de syntaxis van de operatoren gespecificeerd. De tweede operatie is een procedure die het derde argument verandert. Het had ook een functie kunnen zijn die een nieuwe kubus als resultaat aflevert.

Om de specificatie van dit abstracte gegevenstype volledig te maken moeten we eigenlijk ook de semantiek van de operatoren geven. Na de specificatie zouden we met twee dingen verder kunnen gaan. We zouden de rest van het applicatieprogramma kunnen schrijven, dat de kubuspuzzel oplost in termen van de kubusoperaties, of we zouden het abstracte gegevenstype kunnen gaan definiëren. Pas als we met het implementeren van het abstracte gegevenstype beginnen, moeten we de representatie specificeren, en die zou dan best dezelfde driedimensionale array kunnen zijn die we eerder hebben gebruikt. Het komt hierop neer:

in de traditionele aanpak
bepalen we eerst de representatie,
terwijl we in de aanpak met abstracte gegevenstypen
eerst bepalen hoe de gegevens zullen worden gebruikt.

Hoe weten we dat een programma correct is geïmplementeerd? Hoe weten we bijvoorbeeld dat een kubusprogramma werkelijk de kubuspuzzel oplost en niet vals speelt? Neem eens een mens die de kubuspuzzel oplost. Er zijn verschillende methoden. De eerste is het vinden van een serie draaiingen die de kubus in de uitgangsstand brengt. De tweede is de kubus uit elkaar halen en weer op de juiste manier in elkaar zetten. Een derde methode is de gekleurde plastic vierkantjes eraf trekken en weer op de kubus plakken. Alleen de eerste methode lost de puzzel zonder vals spelen op. We zouden kunnen toekijken om er zeker van te zijn dat de proefpersoon die methode gebruikt. Maar hoe kunnen we nagaan of een computer de kubuspuzzel echt oplost? We kunnen het programma natuurlijk een serie draaiingen laten afdrukken die we dan zelf kunnen controleren. Maar die controle is misschien moeilijk uit te voeren als het om een lange serie draaiingen gaat; we zouden ook geen garantie hebben dat de computer ook morgen de puzzel eerlijk oplost, of volgend jaar.

Een andere methode is het bekijken van de structuur van het programma. Als het programma volgens de traditionele aanpak is ontworpen en de representatie op allerlei plaatsen in het programma wordt gebruikt, zouden we het gehele programma moeten bestuderen om tot de overtuiging te komen dat het een eerlijk programma is. Als het programma is ontworpen op basis van het boven gegeven abstracte gegevenstype, dan behoeven we alleen de drie operaties te bestuderen. In feite hoeven we alleen de draaiingsoperatie zorgvuldig te bekijken, want dat is de enige operatie die een verandering in de kubus aanbrengt. Deze methode zou een overtuigender en eenvoudiger bewijs zijn dat het programma eerlijk is.

Iets anders dat we kunnen opmerken over de twee bovenstaande methoden is, dat de aanpak met abstracte gegevenstypen op een natuurlijke manier leidt tot een beter gestructureerd en gemoduleerd programma. De goed gedefinieerde interface die door de operatoren wordt gespecificeerd, leidt tot het onafhankelijk ontwikkelen van twee delen van het programma. Deze heldere structuur geeft de eerste aanpak niet altijd.

Op dit punt is het een redelijke vraag in welk opzicht gegevensabstractie verschilt van procedurele abstractie. Het verschil is dat een gegevensabstractie bestaat uit een verzameling routines die elk toegang hebben tot gemeenschappelijke gegevens of een gemeenschappelijke representatie. De routines zijn hecht met elkaar verweven. We kunnen één losse routine uit deze groep op zichzelf geen abstractie noemen, omdat die routine essentieel afhankelijk is van de correctheid en zelfs van het bestaan van de andere routines in de groep.

We kunnen de belangrijke aspecten van gegevensabstracties als volgt samenvatten:

1. Gegevensabstracties brengen een scheiding aan tussen het gebruik van een gegevenstype en de implementatie ervan.
2. Gegevensabstracties maken correctheidsbeschouwingen eenvoudiger.
3. Gegevensabstracties maken het vervangen van de ene (correcte) implementatie door de andere mogelijk. In beginsel is efficiëntie het enige criterium bij het kiezen van een implementatie.
4. Gegevensabstractie is een techniek voor programmatuurontwerp die modulariteit bevordert en ook de onafhankelijke ontwikkeling van enerzijds de implementatie van de gegevensabstractie en anderzijds het applicatieprogramma.

8.2 Van de class van SIMULA tot de cluster van CLU

Het begrip class in de programmeertaal SIMULA is een belangrijke bron van inspiratie geweest voor het werk aan abstracte gegevenstypen uit de begintijd. SIMULA is ontworpen voor het schrijven van simulatie-programma's. Een voorbeeld in SIMULA-stijl van een class voor het type bag is:

```
class BAG(MAXSIZE: INTEGER) is

        BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
        SIZE: INTEGER := 0;

        INSERT: procedure (X:INTEGER) is
                J: INTEGER;
        begin
                if SIZE ≠ MAXSIZE then
                        SIZE := SIZE + 1;
                        BAG_ELEMENT(SIZE) := X;
                end if;
        end INSERT;

        REMOVE: procedure (X:INTEGER) is
                ...

        IN: function (X:INTEGER) return BOOLEAN is
                ...

end class BAG;
```

De subroutines voor REMOVE en IN zijn hier niet uitgeschreven. Vergelijk dit voorbeeld eens met de operatiedeclaraties die aan het begin van dit hoofdstuk voor BAG zijn gegeven. Het eerste interessante verschil is dat de tweede parameter van elke routine is weggelaten. MAXSIZE is een parameter van de class en geeft aan hoeveel elementen de bag maximaal kan bevatten. Als dat maximum is bereikt, worden verdere toevoegingen genegeerd. Een class-declaratie creëert op zichzelf nog geen objecten. Voor het *genereren* van objecten moet de class worden aangeroepen. Aangezien de class willekeurig vaak kan worden aangeroepen, kunnen er willekeurig veel objecten van het type bag worden gegenereerd. Hier is een voorbeeld van een declaratie die bags genereert en gebruikt:

```
BAG_A: pointer BAG;
VIJF_BAGS: array (1..5) of pointer BAG;

begin
        BAG_A := new BAG(500);
        for J in 1..5 loop
```

```
            VIJF_BAGS(J) := new BAG(100);
        end loop;

        for J in 31..39 loop              -- plaats getallen
            BAG_A.INSERT(J);              -- in BAG_A
        end loop;
        ...
```

In SIMULA gebruikt men het sleutelwoord `ref` in plaats van `pointer`, maar de betekenis van beide sleutelwoorden is voldoende verwant om deze vrijheid te rechtvaardigen. Een aanroep van een class wordt voorafgegaan door het sleutelwoord `new`. Het kan als een allocator worden beschouwd, omdat het geheugen alloceert voor een nieuwe class. Elke class-identifier moet van een nadere aanduiding worden voorzien; elke verwijzing naar een class identifier moet worden voorafgegaan door een bepaalde class. In SIMULA wordt een punt gebruikt als scheiding tussen een class en een identifier van de class. De expressie `VIJF_BAGS(3).REMOVE` verwijst naar de procedure `REMOVE` die hoort bij de derde bag in de array `VIJF_BAGS`. Op dezelfde manier waarop elke class routine toegankelijk is, is ook elke class variabele toegankelijk. In SIMULA zijn class-variabelen van oudsher niet beschermd. Men kan daarom dingen schrijven als:

```
BAG_A.SIZE := 1;
VIJF_BAGS(3).BAG_ELEMENT(4) := 9;
```

Als we het sleutelwoord `class` vervangen door `record`, dan zien we (als we afzien van de parameters van een class en van lokaliteit) dat een class eigenlijk een record is met procedure- en functiecomponenten. Zelfs de syntaxis, met `new` en de puntnotatie, lijkt op die voor records in Pascal of Ada. Er zijn twee verschillen: parameters en lokaliteit. Elke parameter van een class is gewoon een extra veld van het record. Het lokaliteitsprobleem betreft de identifiers in de class routines die naar de class variabelen verwijzen. In de procedure `INSERT` wordt bijvoorbeeld `SIZE` met 1 verhoogd. Als de class gewoon een record is, dan moet bij `SIZE` volledig worden gespecificeerd om welke bag het gaat. Al met al lijkt een class van SIMULA sterk op een record, met twee kleine afwijkingen: class parameters en de bepaling waar een class identifier in een class routine naar verwijst.*

Het probleem waar een class-identifier naar verwijst, leidt tot het volgende. Het is soms nuttig om in een class-routine naar de gehele class te verwijzen. In som-

---

* *Overerving* (*inheritance*), gewoonlijk beschouwd als een onderdeel van het class-concept, wordt apart behandeld in paragraaf 8.5.

mige talen krijgt een class die naar zichzelf verwijst een speciaal sleutelwoord. In Smalltalk is dat sleutelwoord `self`, in SIMULA `this`. Deze voorziening kunnen we gebruiken om het probleem van de betekenis van een class-identifier op te lossen. Met behulp van het sleutelwoord `this` kunnen we de procedure `INSERT` herschrijven tot:

```
INSERT: procedure (X:INTEGER) is
        J: INTEGER;
begin
        if this.SIZE ≠ this.MAXSIZE then
                this.SIZE := this.SIZE + 1;
                this.BAG_ELEMENT(this.SIZE) := X;
        end if;
end INSERT;
```

Als we de procedure herschrijven met de `with` opdracht uit Pascal, krijgt hij bijna weer zijn oude gedaante:

```
INSERT: procedure (X:INTEGER) is
        J: INTEGER;
begin
        with this do begin
                if SIZE ≠ MAXSIZE then
                        SIZE := SIZE + 1;
                        BAG_ELEMENT(SIZE) := X;
                end if;
        end;
end INSERT;
```

Het begrip class van SIMULA is heel nuttig gebleken. Maar voordat dit begrip leidde tot het moderne begrip van abstracte gegevenstypen is er eerst nog een andere belangrijke stap gezet. Voor het eerst werd die gezet in een taal genaamd CLU, waar de betreffende constructie de naam cluster kreeg. Een cluster is een SIMULA-class met privé variabelen. Om de problemen te voorkomen die in SIMULA bestaan met class-identifiers en `this`, geven de cluster-routines in CLU alle parameters expliciet door. Er wordt voor protectie gezorgd doordat er onderscheid wordt gemaakt tussen de interne en de externe representatie van de gegevens. Alleen de cluster kent de interne representatie. Buiten de cluster wordt het type alleen een naam gegeven en kunnen alleen de zichtbare operaties van de cluster voor gegevens van het betreffende type worden gebruikt.

8.3 Programmeren met abstracte gegevenstypen in talen zonder abstracte gegevenstypen

In deze paragraaf onderzoeken we een aantal uiteenlopende methoden om het concept van abstracte gegevenstypen te gebruiken in talen die hier geen speciale faciliteiten voor bieden. Hierbij zal duidelijk worden dat een abstract gegevenstype evenzeer een ontwerptechniek is als een taalconstructie. De meeste programmatuur is geschreven in talen die geen voorzieningen hebben voor abstracte gegevenstypen; maar het gebruik van zo'n taal is geen reden om abstracte gegevenstypen te vermijden. Hoewel het opbouwen van gegevensabstracties in die talen moeilijkheden ondervindt, zoals het ontbreken van automatische controle, leidt het toch tot een kwalitatief beter programmatuurprodukt.

Bij het implementeren van abstracte gegevenstypen moeten we een aantal belangrijke beslissingen nemen ten aanzien van de stijl waarin dat gebeurt. Het betreft hier punten als de mate van formaliteit die wordt gebruikt, de wijze waarop procedures worden aangeroepen, kwesties van representatie en natuurlijk de hoeveelheid protectie die nog uit de taal kan worden geperst.

1. Moet de implementatie maar één of vele objecten van het gewenste gegevenstype toestaan?
2. Kan de implementatie op één bepaalde plaats in elkaar worden gezet, afzonderlijk van de rest van het programma?
3. Moet de implementatie waarde-gericht of variabele-gericht zijn?
4. Moet de implementatie een gedeelde toekenning of een waardetoekenning (zie paragraaf 7.3) hebben (of misschien helemaal geen)?
5. Wat voor soort protectie kan er worden geleverd? En welke willen we leveren?
6. Wordt de implementatie onafhankelijk gecompileerd?
7. Worden generieke, dus type-onafhankelijke operaties (zoals vergelijking en toekenning) toegestaan? Zo niet, wordt er een vervangende methode aangeboden?
8. Komen er wel generieke faciliteiten?

Een of vele variabelen

Er bestaat een eenvoudige en duidelijke manier om abstracte gegevenstypen te implementeren in talen die ze niet hebben, maar daarbij treedt wel een verlies van algemeenheid op. Die methode gaat ervan uit dat er maar één exemplaar

van het betreffende gegevenstype wordt gebruikt. De programmeertaal die voor de implementatie wordt gebruikt, moet in staat zijn een verzameling routines te compileren (of één routine met meerdere ingangen), met statische variabelen die alleen voor die routines toegankelijk zijn. Onder die omstandigheden kunnen abstracte gegevenstypen waarvan maar één exemplaar wordt gebruikt, zeer economisch worden geïmplementeerd; er treden in dat geval ook maar weinig problemen op met de andere kwesties die hierboven zijn opgesomd. Maar meestal zijn er meerdere variabelen van een abstract gegevenstype gewenst en als we die gelegenheid willen geven, krijgen we te maken met vele van de problemen die we hierna zullen bespreken. Om enkele van deze punten te illustreren blijven we gebruik maken van het abstracte type bag. Het volgende programmasegment laat zien hoe we een enkele variabele van het type bag zouden kunnen implementeren.

```
MAXSIZE: constant := 100;
BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
SIZE: INTEGER := 0;
INSERT: procedure (X:INTEGER) is
        J: INTEGER;
begin
        if SIZE ≠ MAXSIZE then
                SIZE := SIZE + 1;
                BAG_ELEMENT(SIZE) := X;
        end if;
end INSERT;

REMOVE: procedure (X:INTEGER) is
        ...

IN: function (X:INTEGER) return BOOLEAN is
        ...
```

In dit voorbeeld is er maar één bag. Daarom is er in de operaties geen parameter van het type bag opgenomen. Om dezelfde reden is er geen toekenning en zijn er geen vergelijkingsoperaties op bags nodig.

Concentratie op één plaats van het abstracte gegevenstype

We gebruiken de *package* van Ada om dingen op één plaats bij elkaar te zetten omdat ze logisch met elkaar verbonden zijn. In het voorafgaande voorbeeld zijn de variabelen en de procedures allemaal logisch met elkaar verbonden. Om het programma beter te organiseren en gemakkelijker leesbaar te maken, is het belangrijk deze zaken te groeperen. In andere talen is dit groeperen meestal ook mogelijk. Het is zelfs beter om het abstracte gegevenstype apart in een bestand

te plaatsen en afzonderlijk te compileren. In PL/I is het mogelijk één routine te construeren met meerdere ingangen, één ingang per operatie. Binnen de routine kunnen gegevens in PL/I als statisch worden gedeclareerd. Pascal kan niet worden gebruikt om declaraties te groeperen die met elkaar verbonden zijn. In Pascal moeten declaraties in een bepaalde volgorde staan. Eerst moeten alle labels voorkomen, dan de constanten, de typedeclaraties, de variabelen en de routines. De volgorde van deze declaraties mag niet worden veranderd. Als gevolg daarvan moeten de zaken die samen een abstract gegevenstype bepalen, verspreid staan tussen de andere declaraties en routines. Ze kunnen niet, apart van de andere declaraties, op één plaats bij elkaar worden gezet. Dat feit maakt het erg moeilijk om in Pascal abstracte gegevenstypen vorm te geven. In sommige moderne Pascal-compilers is afzonderlijke compilatie mogelijk. Bij die compilers kunnen abstracte gegevenstypen in aparte bestanden worden geplaatst. Van nu af zullen we in voorbeelden het sleutelwoord `adt` (abstract data type) gebruiken voor het groeperen:

```
adt BAGS is
          ...
end adt BAGS;
```

Waarde-gerichte en variabele-gerichte abstracte gegevenstypen

In een *waarde-gerichte* implementatie van een abstract gegevenstype wordt het gegevenstype uitsluitend als een verzameling waarden en een verzameling operaties op die waarden gezien. Kwesties van variabelen en geheugenopslag worden niet in het abstracte gegevenstype betrokken. In een *variabele-gerichte* implementatie van een abstract gegevenstype wordt het gegevenstype gezien als een verzameling variabelen en waarden, waarbij sommige van de operaties op variabelen van het betreffende gegevenstype werken. Een waarde-gerichte implementatie is misschien handiger en esthetischer, maar variabele-gerichte implementaties zijn vaak efficiënter. Vergelijk de volgende twee implementaties van het abstracte gegevenstype bag.

```
adt BAGS is  -- variabele-gerichte implementatie
    MAXSIZE: constant := 100;
    type BAG is
        record
            BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
            SIZE: INTEGER := 0;
        end record;
```

```
    INSERT: procedure (X:INTEGER, B: in out BAG) is
    begin
        if B.SIZE ≠ MAXSIZE then
           B.SIZE := B.SIZE + 1;
           B.BAG_ELEMENT(B.SIZE) := X;
        end if;
    end INSERT;

    REMOVE: procedure (X:INTEGER, B: in out BAG) is
        ...

    IN: function (X:INTEGER, B:BAG) return BOOLEAN is
        ...

end adt BAGS;

adt BAGS is              -- waarde-gerichte implementatie
    MAXSIZE: constant := 100;
    type BAG is
        record
            BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
            SIZE: INTEGER := 0;
        end record;

    INSERT: procedure (X:INTEGER, B:BAG) return BAG is
        RESULT: BAG := B;
    begin
        if RESULT.SIZE ≠ MAXSIZE then
           RESULT.SIZE := RESULT.SIZE + 1;
           RESULT.BAG_ELEMENT( RESULT.SIZE) := X;
        end if;
        return RESULT;
    end INSERT;

    REMOVE: procedure (X:INTEGER, B:BAG) return BAG is
        ...

    IN: function (X:INTEGER, B:BAG) return BOOLEAN is
        ...

end adt BAGS;
```

In de variabele-gerichte implementatie worden bag-variabelen aan procedures of functies doorgegeven en rechtstreeks gewijzigd. In de waarde-gerichte implementatie worden bag-waarden aan functies doorgegeven die nieuwe bag-waarden creëren. Variabele-gerichte implementaties zijn efficiënter, maar kunnen zij-effecten hebben. In het gebruik zien deze verschillende implementaties er ook verschillend uit.

```
X, Y, Z: BAG;                          -- variabele-gerichte bags
INSERT(5, X);
REMOVE(8, Y);
```

```
X, Y, Z: BAG;                          -- waarde-gerichte bags
X := INSERT(5, X);
Y := REMOVE(8, Y);
```

In beide versies hierboven is het initialiseren van een bag-waarde een probleem. Het is van belang om de SIZE van een bag op nul te initialiseren. Dat is in de twee bovenstaande voorbeelden gemakkelijk te doen. Maar in vele programmeertalen is het niet mogelijk een component van een record-type in een typedeclaratie te initialiseren. Eén van de manieren om het initialisatieprobleem op te lossen is het maken van een initialisatiefunctie. In de variabele-gerichte implementatie zou die er zo kunnen uitzien:

```
LEEG_BAG: procedure (B: in out BAG) is
begin
        B.SIZE:=0;
end LEEG_BAG;
```

In implementaties die maar één variabele per abstract gegevenstype toelaten, is het eenvoudig de variabele, de opslag in het geheugen en de initialisatie volledig in de hand te houden. Helaas bieden de meeste programmeertalen dergelijke faciliteiten niet voor implementaties voor meerdere variabelen.

Protectie

Eén van de belangrijkste problemen bij het implementeren van abstracte gegevenstypen in talen zonder voorzieningen hiervoor is, dat het niet mogelijk is te controleren of de gegevensabstracties op de juiste wijze worden gebruikt. In een taal zonder zulke faciliteiten moet de protectie ad hoc zijn en wordt er meestal van uitgegaan dat iedereen competent en goedwillend is. In een taal met ingebouwde abstracte gegevenstypen voert de compiler die controle uit. In Ada kan het sleutelwoord `private` worden gebruikt om voor de nodige protectie te zorgen; als dat woord wordt gebruikt, betekent het dat de representatie buiten de gegevensabstractie niet bereikbaar is. In een implementatie die zich tot één enkele variabele beperkt, kunnen de gegevens voldoende worden beschermd als de taal de juiste regels hanteert voor het geldigheidsgebied van identifiers. In PL/I houdt bijvoorbeeld een abstract gegevenstype dat is geïmplementeerd als een routine met meerdere ingangen, al zijn statische variabelen voor de buitenwereld verborgen. In een taal met afzonderlijke compilatie kan men variabelen vaak in een apart bestand verborgen houden door ze niet als extern te declareren. In het geval van de taal C kan een variabele als statisch worden gedeclareerd, hetgeen betekent dat de variabele in dat bestand bekend is, maar daarbuiten niet.

In talen zonder voorzieningen voor gegevensabstractie kunnen ongetypeerde pointers soms voor enige bescherming zorgen. In PL/I hebben pointers geen type en daarom is het in een programma waarin een abstract gegevenstype wordt gebruikt, niet nodig de representatie te specificeren. Hoewel in C pointers wel een type hebben, is het gemakkelijk een onvolledige specificatie te geven van datgene waarnaar de pointer in feite wijst; dat vermindert de kans op misbruik, doordat de representatie niet onmiddellijk bij de hand is. Helaas kan men in beide talen, als men de representatie kent, daar gebruik van maken, uiteraard zonder dat de compiler daartegen waarschuwt.

Het volgende programma is een voorbeeld van het gebruik van pointers om de representatie verborgen te houden.

```
adt BAGS is  -- variabele-gerichte implementatie met pointers
    MAXSIZE: constant := 100;
    type BAG is pointer BAGREP;
    type BAGREP is
        record
            BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
            SIZE: INTEGER := 0;
        end record;

    INSERT: procedure (X:INTEGER, B: in out BAG) is
    begin
        if B.SIZE ≠ MAXSIZE then
            B.SIZE := B.SIZE + 1;
            B.BAG_ELEMENT(B.SIZE) := X;
        end if;
    end INSERT;

    REMOVE: procedure (X:INTEGER, B: in out BAG) is
        ...

    IN: function (X:INTEGER, B:BAG) return BOOLEAN is
        ...

end adt BAGS;
```

Bovenstaand programma kan onafhankelijk worden gecompileerd. Alleen de declaraties van BAG, INSERT, REMOVE en IN worden aan de gebruiker van het abstracte gegevenstype bag bekend gemaakt. De declaratie van BAGREP en de inhoud van de procedures worden geheim gehouden. Zowel in C als in PL/I is dat gemakkelijk te doen. Het gebruik van pointers voor het representeren van abstracte gegevenstypen heeft nog andere voordelen. Pointers kunnen altijd worden gebruikt om bij het aanroepen van een functie waarden door te geven en als resultaat af te leveren. Sommige talen staan het doorgeven en als resultaat afle-

veren van bepaalde typen niet toe. Het is ook efficiënter om bij toekenningen, parameteroverdracht en het afleveren van het resultaat van een functie pointers te kopiëren. Maar er zijn ook problemen. De programmeur heeft nu de zorg voor het alloceren en saneren van geheugenruimte. Ook kan initialisatie moeilijker zijn. De toekenning wordt een gedeelde toekenning, iets wat voor de onoplettende gebruiker onverwacht kan zijn.

Een iets veiliger methode is het vervangen van een pointer door een index in een verborgen array. Die index kan op dezelfde manier worden gebruikt als een pointer. Het verschil met de vorige methode is, dat zelfs als de representatie bekend is, de plaats van de array nog niet bekend is. Zo'n implementatie zou er zo kunnen uitzien:

```
adt BAGS is  -- variabele-gerichte implementatie met indices
    MAXSIZE: constant := 100;
    type BAG is 1..100;
    ALLBAGS: array (BAG) of
        record
            BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
            SIZE: INTEGER := 0;
        end record;

    INSERT: procedure (X:INTEGER, B: in out BAG) is
        J: INTEGER;
    begin
        if ALLBAGS(B).SIZE ≠ MAXSIZE then
            ALLBAGS(B).SIZE := ALLBAGS(B).SIZE + 1;
            J:= ALLBAGS(B).SIZE;
            ALLBAGS(B).BAG_ELEMENT(J) := X;
        end if;
    end INSERT;

    REMOVE: procedure (X:INTEGER, B: in out BAG) is
        ...

    IN: function (X:INTEGER, B:BAG) return BOOLEAN is
        ...

end adt BAGS;
```

Voorgedefinieerde operaties

Sommige operaties, zoals toekenning en vergelijking, kunnen in de meeste talen op alle gegevenstypen worden uitgevoerd. Deze operaties moeten ook in beschouwing worden genomen bij het opbouwen van nieuwe gegevenstypen.

In de meeste gevallen loopt dat vanzelf goed en doen ze wat van ze wordt verwacht, maar in sommige gevallen doen ze iets onverwachts. In Ada is er een aantal mechanismen dat voor het oplossen van zulke problemen kan worden gebruikt, maar in de meeste talen kunnen ze niet worden opgelost.

Laten we eerst eens kijken naar de toekenning. Bij het gebruik van pointers of indices wordt de toekenningsoperatie een gedeelde toekenning. Dat kan tot onverwacht gedrag leiden. Beschouw het volgende stuk code:

```
X, Y, Z: BAG;

begin
        X:=Y;
        INSERT(6, Y);
```

Men is geneigd de toekenning als een kopieeroperatie te zien en te denken dat het toevoegen van 6 aan `Y` de waarde van `X` niet aantast. Maar als bags worden gerepresenteerd als pointers of indices, kopieert de toekenning alleen de pointer of de index, niet de waarde. Maar in talen als C en PL/I is de prijs voor het niet gebruiken van pointers hoog. Hier moet dan een zorgvuldige afweging worden gemaakt. Men kan als algemene beleidslijn kiezen dat gegevensabstracties pointers en gedeelde toekenningen gebruiken en dat de gebruikers maar moeten opletten. Een manier om het probleem te verkleinen is het schrijven van een routine `COPIEER` of `KEN_TOE` die bij het gebruik van de toekenningsopdracht als hulp dient of die zelfs vervangt. De toekenningsroutine zou eruit kunnen zien als:

```
KEN_TOE: procedure (LINKS: in out BAG; RECHTS: BAG) is
begin
        LINKS.SIZE          := RECHTS.SIZE;
        LINKS.BAG_ELEMENT := RECHTS.BAG_ELEMENT;
end KEN_TOE;
```

Het andere ernstige probleem treedt op bij vergelijking. In de meeste talen mogen van elk type de waarden worden vergeleken. Beschouw het volgende stuk code:

```
X, Y, Z: BAG;

begin
        INSERT(5, X);
        INSERT(6, X);
        INSERT(6, Y);
        INSERT(5, Y);
        if X=Y then ...
```

In de laatste opdracht wordt de voorgedefinieerde gelijkheidsoperator gebruikt. In de variabele-gerichte implementaties die we hierboven hebben besproken zouden de eerste vier procedure-aanroepen aan SIZE de waarde 2 geven, aan de eerste twee elementen van X.BAG_ELEMENT de waarden 5 en 6 en aan de eerste twee elementen van Y.BAG_ELEMENT de waarden 6 en 5. Als de representatie met behulp van pointers of indices plaatsvindt, worden er pointers of indices vergeleken. Als de representatie een record is, worden de velden SIZE en BAG_ELEMENT vergeleken. In beide gevallen is het resultaat false. In werkelijkheid zou men willen dat de vergelijking true oplevert, aangezien X en Y dezelfde bag representeren, namelijk een bag die de waarden 5 en 6 bevat. In welke volgorde de waarden in de bag zijn gestopt, doet er niet toe. In het algemeen geldt dat een waarde van een gegevenstype geen unieke representatie hoeft te hebben. Zelfs getallen kunnen op verschillende manieren worden gerepresenteerd. In de één-complement-notatie zijn er twee representaties voor nul. Net als voor de toekenning kan men ook een functie GELIJK schrijven als vervanging van de gelijkheidsoperator.

8.4 Ada-packages

De Ada-package wordt gebruikt om een verzameling declaraties en procedures te groeperen. Er is daarin zowel plaats voor privé als voor openbare declaraties en afzonderlijke compilatie is mogelijk. Zo'n package kan worden gebruikt om abstracte gegevenstypen te implementeren. Het abstracte gegevenstype bag kan met behulp van Ada-packages op verscheidene manieren worden geïmplementeerd, waaronder de methoden uit de vorige paragraaf. Hier volgt een voorbeeld van het gebruik van een package ten behoeve van afzonderlijke compilatie:

```
package BAGS is  -- variabele-gerichte implementatie
    MAXSIZE: constant := 100;
    type BAG is private;
    procedure INSERT(X:INTEGER, B: in out BAG);
    procedure REMOVE(X:INTEGER, B: in out BAG);
    function IN(X:INTEGER, B:BAG) return BOOLEAN;

private

    type BAG is
        record
            BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
            SIZE: INTEGER := 0;
        end record;

end BAGS;
```

```
package body BAGS is

    procedure INSERT(X:INTEGER, B: in out BAG) is
    begin
        if B.SIZE ≠ MAXSIZE then
           B.SIZE := B.SIZE + 1;
           B.BAG_ELEMENT(B.SIZE) := X;
        end if;
    end INSERT;

    procedure REMOVE(X:INTEGER, B: in out BAG) is
        ...

    function IN(X:INTEGER, B:BAG) return BOOLEAN is
        ...

end BAGS;
```

Het abstracte gegevenstype is hier in twee delen gesplitst. Het zichtbare deel staat bovenaan. De declaraties tot aan het los op de regel staande sleutelwoord `private` zijn buiten de package zichtbaar. Het tweede deel wordt *package body* genoemd en kan afzonderlijk worden gecompileerd. Merk op dat de syntaxis en de representatie deel zijn van de package en de algoritmen in het package body staan. De semantiek van het abstracte gegevenstype komt helemaal niet voor.

Het lijkt misschien vreemd dat de representatie van het abstracte gegevenstype in de package in plaats van in het package body optreedt. Het zou ideaal zijn als de representatie in het package body zou staan, zodat het ene package body gemakkelijk door het andere kan worden vervangen. De representatie maakt deel uit van de implementatie en zal waarschijnlijk van implementatie tot implementatie verschillen. Als de representatie in de package staat in plaats van in het package body, dan maakt een verandering in het package body (in de meeste gevallen) ook een verandering in de package zelf noodzakelijk. De programma's die de package gebruiken zullen daardoor ook opnieuw moeten worden gecompileerd. Deze ongelukkige eigenschap is te wijten aan enkele van de doelen die bij het ontwerpen van Ada een hogere prioriteit hadden. Die ontwerpdoelen waren onder andere:

1. het tijdens het compileren vaststellen van de vereiste hoeveelheden geheugen;
2. de onafhankelijke compilatie van package en package body.

Om de hoeveelheid geheugen vast te stellen die nodig is voor het opbergen van de objecten van het abstracte gegevenstype, is het nodig de representatie te ken-

nen. Om die reden wordt de representatie opgenomen in de package in plaats van in het package body. Om de representatie privé te laten zijn moet de gebruiker het sleutelwoord `private` gebruiken. De representatie behoeft niet geheim te blijven, aangezien de compiler het juiste gebruik afdwingt.

8.5 Subtypen en typehiërarchieën

In het meest algemene geval is een typehiërarchie een partiële ordening op gegevenstypen. Subtype en supertype zijn termen om deze ordening aan te geven. Het type A is een *subtype* van B (of B is een *supertype* van A) als A lager geplaatst of kleiner is dan B. Deze ordening of hiërarchie wordt op verschillende criteria gebaseerd en er bestaat geen algemeen aanvaarde standaarddefinitie. In deze paragraaf zullen we gewoon verschillende standpunten weergeven. In het algemeen gaat het bij een hiërarchie om eigenschappen die gegevenstypen gemeenschappelijk hebben, met name operaties en/of waarden. Twee globale mogelijkheden zijn de volgende. A wordt als subtype van B beschouwd als alle waarden van het type A ook waarden van het type B zijn. Of A wordt als subtype van B beschouwd als alle operaties op type B ook beschikbaar zijn als operaties op het type A. Men zegt dan dat A de operaties van B *erft*, omdat B vaak het *oudertype* wordt genoemd.

We beginnen met een paar mogelijke hiërarchieën die niet zo vaak worden genoemd. De eerste is op het type union gebaseerd. De typen A en B zijn subtypen van de union van A en B. Dit schema genereert een partiële ordening op de typen, maar die is meestal niet wat met een hiërarchie wordt bedoeld. Dat wil zeggen, het subtype bevat geen van de operaties en de waarden van het supertype, omdat de waarden van een union van een speciaal kenmerk zijn voorzien en alleen met behulp van een projectie-operatie kunnen worden veranderd in een waarde van één van de samenstellende typen. Polymorfe typen (zie hoofdstuk 11) leveren een andere mogelijke hiërarchie. Hierbij zijn array van integer en array van teken subtypen van het type array. Maar deze hiërarchie gaat ervan uit dat polymorfe typen, zoals array, op zichzelf staande typen zijn. Dat is meestal niet het geval, omdat polymorfe typen verzamelingen van typen zijn in plaats van typen. In sommige talen, waarin typering dynamisch geschiedt, zoals SNOBOL en SETL, komt zo'n hiërarchie wel in aanmerking omdat in deze talen heterogene arrays toegestaan zijn; zo kan men aan verschillende subtypen denken -- bijvoorbeeld arrays met alleen integers of tekens, arrays met alleen tekens, of arrays met alleen alfabetische tekens. Maar in deze talen bestaat geen goed ontwikkeld begrip 'compile-time type' en de typecontrole wordt grotendeels ge-

daan tijdens de uitvoering van het programma. In zulke talen zijn niet subtypen belangrijk, maar optimalisatie door middel van het afleiden van typen tijdens de compilatie.

In Pascal wordt met het woord subtype een deelbereik bedoeld. De waarden van een type dat een deelbereik is, vormen een deelverzameling van de waarden van een ander type. Beschouw de volgende deelbereiken:

```
type KLEUR is (BLAUW, GROEN, GEEL, ORANJE, ROOD, PAARS);
type WARM is GEEL..ROOD;
type HEET is ORANJE..ROOD;
```

HEET is een deelbereik van WARM, dat weer een deelbereik van KLEUR is. Deelbereiken worden in Pascal niet echt als typen beschouwd. Zo is er bijvoorbeeld tijdens het compileren (en soms zelfs tijdens het uitvoeren van het programma) geen typecontrole voor deelbereiken. Deze opvatting van een subtype is in Ada uitgebreid tot het idee van beperkte typen. Bij een *beperkt* (*constrained*) type worden er grenzen gesteld aan de waarden die een variabele mag aannemen. Voorbeelden van beperkte typen in Ada zijn deelbereiken, arrays met opgegeven grenzen, en variante records die tot een bepaalde variant worden beperkt.

In een andere opvatting die soms wordt gekozen, wordt de type-hiërarchie bepaald door de coërcies van de taal. Als bijvoorbeeld integers impliciet worden geconverteerd naar reals, dan worden integers als een subtype van reals beschouwd. De conversies maken het mogelijk te zeggen dat de integers de operaties van de reals erven. Zo is een vierkantsworteloperatie die op reals is gedefinieerd, ook voor integer parameters beschikbaar, maar denk erom dat het resultaat dan een real waarde is, niet een integer waarde. Merk ook op dat vanuit dit oogpunt een taal met de uniting coërcie (die een waarde van type T converteert naar een waarde van een union-type waar T deel van uitmaakt) een hiërarchie krijgt die verwant is aan de union-hiërarchie die we boven gezien hebben.

Indien gegevenstypen worden beschouwd als verzamelingen operaties op een universeel domein (zoals in Russell), dan kunnen we het begrip subtype definiëren door gewoon naar de beschikbare operaties te kijken. Type A wordt als een subtype van type B beschouwd als de verzameling van operaties van type B een deelverzameling is van de verzameling van operaties van type A. Een type LIJST met operaties voeg_voor, voeg_achter, eerste, rest en leeg is een subtype van QUEUE met operaties voeg_voor, eerste, rest en leeg. De terminologie kan verwarrend werken, want intuïtief lijkt het type QUEUE misschien een subtype van LIJST, omdat het een beperktere verzameling operaties heeft. Maar naar het gangbare gebruik van de term betekent *subtype* het tegengestelde.

De bekendste interpretatie van de term subtype is ontwikkeld in de taal Smalltalk. De Smalltalk-class is gebaseerd op de SIMULA-class en voor ons doel kunnen we ervan uitgaan dat classes gewoon gegevensabstracties zijn. Een class kan worden uitgebreid tot een subclass (ongeveer zoals dat in SIMULA met een prefix gebeurt). De subclass erft alle operaties van de class en daar kan dan nog een eigen verzameling operaties bijkomen. Elke class is een object in een andere class en alle classes zijn objecten van de universele class *Object*, waartoe zulke algemene operaties behoren als toekenning, vergelijking en uitvoer. Een subclass kan één of meer van de geërfde operaties ongeldig verklaren. Beschouw de volgende class bag:

```
class BAG is

    BAG_ELEMENT: array (1..MAXSIZE) of INTEGER;
    SIZE: INTEGER := 0;

    INSERT: procedure (X: INTEGER) is ... end INSERT;

    REMOVE: procedure (X: INTEGER) is ... end REMOVE;

    IN: function (X: INTEGER) return BOOLEAN ... end IN;

    function "=" (B: BAG) return BOOLEAN ... end;

end class BAG;
```

In dit voorbeeld maakt de gelijkheidsoperator die van de superclass *Object* ongeldig en treedt ervoor in de plaats. Die voorziening is nodig omdat het type bag verschillende representaties kent voor dezelfde waarde en de generieke gelijkheidsoperatie niet overeen zou komen met de verwachte interpretatie van gelijkheid van bags.

Laten we nu eens kijken naar een subclass GROTE_BAG, waarin de class BAG wordt uitgebreid met een operatie genaamd INSERT_VELE, die een onuitputtelijk aantal integers in de bag stopt. Voor het opbergen van onuitputtelijke integers zijn nieuwe gegevensstructuren nodig en de routines voor IN en gelijkheid moeten worden vervangen om met de onuitputtelijke integers rekening te houden. De subclass GROTE_BAG erft de operaties INSERT en REMOVE van de class BAG en de toekennings- en de uitvoeroperaties van de class *Object*.

```
class GROTE_BAG is BAG plus

    CBAG: array (1..MAXSIZE) of INTEGER;
    CSIZE: INTEGER := 0;
```

```
        INSERT_VELE: procedure (X: INTEGER) is ... end;

        IN: function (X: INTEGER) return BOOLEAN ... end IN;

        function "=" (B: BAG) return BOOLEAN ... end;

    end class GROTE_BAG;
```

De aanpak van Smalltalk maakt het uitbreiden van record-achtige gegevensstructuren gemakkelijk. Een object PERSOON kan bijvoorbeeld informatie bevatten over de naam van een persoon, zijn geboortedatum en zijn adres. Dit object kan op verschillende manieren worden uitgebreid tot bepaalde subtypen. Een object STUDENT zou een subclass van PERSOON kunnen zijn met extra velden voor jaar, hoofdvak enzovoort. Een KEUZE_STUDENT zou een subclass van STUDENT kunnen zijn met een veld voor het gekozen bijvak.

Merk op dat de waarden van een subclass in Smalltalk niet noodzakelijkerwijs een deelverzameling vormen van de waarden van de ouder-class. Een andere opvatting van subtype met die eigenschap gaat uit van predikaten. Een unaire Booleaanse functie op een type verdeelt de waarden van dat type in twee delen; elk deel kan als een subtype worden beschouwd. De functie IS_LETTER op tekens definieert bijvoorbeeld het alfabetische (en het niet-alfabetische) subtype van het type teken. Het subtype LETTER kan worden gedefinieerd met behulp van de notatie die door Burton en Lings (1981) is geïntroduceerd:

```
IS_LETTER: function (C: CHARACTER) return BOOLEAN is ...

subtype LETTER is CHARACTER which is IS_LETTER;
```

Met deze faciliteit zou in Ada het begrip subtype kunnen worden uitgebreid naar willekeurige verzamelingen van waarden.

Natuurlijk blijft het probleem bestaan of zulke subtypen echte typen zijn, die tijdens het compileren gecontroleerd kunnen en moeten worden. De operaties op zulke subtypen zijn onduidelijk. Beschouw de integers als een subtype van de reals en kijk dan eens naar het erven van de vermenigvuldiging en de deling. Veranderen de typen van de geërfde operaties ook? Blijft bijvoorbeeld de vermenigvuldiging een operatie op twee reals die een real als resultaat heeft, en worden de integers naar reals geconverteerd vóór het aanroepen van de functie? Als dat het gekozen standpunt is, zijn subtypen gewoon een manier om coërcies te definiëren. In plaats daarvan zou er ook een nieuwe vermenigvuldigingsoperatie in het leven kunnen worden geroepen. Moet in dat geval de nieuwe verme-

nigvuldigingsoperatie een real of een integer afleveren? Als het subtype gesloten is onder de operatie (bijvoorbeeld de vermenigvuldiging), kan er veilig een waarde van het subtype worden afgeleverd. Maar als het gaat om een operatie waaronder het subtype niet gesloten is (bijvoorbeeld deling), dan moet er een real getal worden afgeleverd.

Een andere opvatting van subtypen berust op kwesties van implementatie. Men zou een hiërarchie van verzamelingstypen kunnen wensen. De algemeenste soort verzameling zou een inefficiënte implementatie kunnen hebben, maar wel rekening houden met oneindige verzamelingen. Subtypen als eindige verzamelingen, reguliere verzamelingen, verzamelingen van beperkte grootte (minder dan *N* elementen) en verzamelingen tekens kunnen op een andere manier zijn geïmplementeerd, veel efficiënter dan de meest algemene soort. Aan de andere kant wil men al deze verschillende soorten verzamelingen waarschijnlijk niet als verschillende typen zien, maar liever als subtypen van één supertype. De subclass van Smalltalk is voor dit doel niet geschikt, omdat daarbij een gemeenschappelijke representatie van de gegevens wordt verondersteld.

Opgaven

1. Ontwerp en implementeer bags met de volgende representatie:

```
array (CHAR) of INTEGER;
```

   Elk element van de array geeft het aantal keren aan dat het teken in de bag voorkomt. Als de bag leeg is zijn alle elementen nul. Bespreek de verschillen in efficiëntie tussen deze implementatie en de implementaties die in dit hoofdstuk zijn beschreven.

2. Beschouw het volgende abstracte gegevenstype functie:

```
adt FUNC_PAKKET is
      type FUNC is private;

      DEFINIEER: function (F: function (X:INTEGER)
                                return INTEGER)
         return FUNC;

      PAS_TOE: function (FN: FUNC; X:INTEGER)
                  return INTEGER;

end adt FUNC_PAKKET;
```

De functie DEFINIEER initialiseert een functie en PAS_TOE past die functie toe op een argument. Oftewel: voor elke integer X en elke unaire integer functie F geldt altijd:

```
PAS_TOE(DEFINIEER(F), X) = F(X)
```

Implementeer dit gegevenstype op zo'n manier dat de berekende waarden van de functie F worden bewaard. De eerste keer dat F met een bepaald argument wordt aangeroepen, wordt de berekende waarde op een of andere lijst bewaard. Latere aanroepen van F met hetzelfde integer argument leveren dan de bewaarde waarde af zonder F nog eens aan te roepen. Zo'n implementatie vereist dat F zich gedraagt als een wiskundige functie: geen zij-effecten, geen historie en geen afhankelijkheid van globale variabelen. In gevallen waarin een enkele aanroep van F veel tijd kost en F vaak met dezelfde argumenten wordt aangeroepen, verkort deze implementatie de tijd die gemiddeld voor een aanroep nodig is.

3. Bestudeer de mogelijkheid om het abstracte gegevenstype verzameling in een of andere taal te implementeren (kies een taal als PL/I, FORTRAN, Pascal, C of BASIC). Bespreek de voor- en nadelen van die implementatie. Ga de lijst van punten af die aan het begin van paragraaf 8.3 is gegeven.

Literatuur

Dahl en Hoare (1972) onderzoeken het gebruik van SIMULA-classes voor abstracte gegevenstypen en coroutines. Er zijn verscheidene programmeertalen bedacht met faciliteiten voor gegevensabstractie; enkele van de bekendste zijn:

CLU	Liskov et al. (1977)
Alphard	Wulf et al. (1976)
Mesa	Geschke et al. (1977)
Euclid	Lampson et al. (1977)
Ada	Ichbiah et al. (1979)
Modula-2	Wirth (1980)
Smalltalk	Goldberg en Robson (1983)

Stroustrup (1986) heeft een uitbreiding van de taal C voor gegevensabstractie onderzocht. Onderzoeksrichtingen op het gebied van gegevensabstracties worden besproken door Shaw (1976). Een bibliografie op het terrein van gegevens-

typen en gegevensabstracties is te vinden in Dungan (1979). Dewar et al. (1979) laten een andere manier zien om implementatie en gebruik van een gegevenstype in SETL te scheiden. Typehiërarchieën en subtypen worden beschreven in Burton en Lings (1981), Albano (1983), Sherman (1984) en Goldberg et al. (1983). Het gebruik van verscheidene implementaties voor één gegevensabstractie wordt besproken door White (1983) en Sherman (1984).

9

Voorbeelden van abstracte gegevenstypen

Dit hoofdstuk bevat een aantal grotere voorbeelden van gegevensabstracties. In het vorige hoofdstuk stonden kleinere voorbeelden die verschillende technieken en kwesties illustreerden, maar die niet het belang konden tonen dat grotere voorbeelden van gegevensabstracties kunnen laten zien. In dit voorbeeld tonen gegevensabstracties die opgebouwd zijn met behulp van andere gegevensabstracties, hoe grotere gegevensabstracties er in de praktijk uit kunnen zien.

9.1 Tekenverzamelingen

Een tekenverzameling is een goed voorbeeld van het gebruik van gegevensabstracties, omdat tekenverzamelingen voorkomen in vele verschillende implementaties waarmee iedereen vertrouwd is. Het totale domein (dat wil zeggen de verzameling van alle mogelijke elementen) is de verzameling van alle tekens, zoals de tekenverzamelingen ASCII en EBCDIC. De operaties van onze tekenverzameling zullen zijn:

lege_set( )	levert de lege verzameling af
voeg_toe(c, s)	levert de vereniging af van s en {c}
verwijder(c,s)	levert s–{c}

vereniging(s1,s2)	levert de vereniging af van s1 en s2
lid(c,s)	levert TRUE af als c tot s behoort, anders FALSE.

We gaan uit van een waardegerichte implementatie zodat de gegevensabstractie van de tekenverzameling er als volgt uitziet:

```
adt CHAR_SET is
    type CHARSET is private;

    LEGE_SET:    function                          return CHARSET;
    VOEG_TOE:    function (C: CHAR; S: CHARSET)    return CHARSET;
    VERWIJDER:   function (C: CHAR; S: CHARSET)    return CHARSET;
    VERENIGING:  function (VERZ1, VERZ2: CHARSET)return CHARSET;
    LID:         function (C: CHAR; S: CHARSET)    return BOOLEAN;

end adt CHAR_SET;
```

Voor verzamelingen bestaan er twee klassieke implementaties: een geketende lijst en een bit-map. De eerste implementatie plaatst de elementen van de verzameling in een geketende lijst. De functie 'lege_set' levert dan de lege lijst af. De functie 'voeg_toe' voegt een element aan de lijst toe, en de functie 'verwijder' haalt er een weg. Bij een waarde-gerichte implementatie vereisen deze veranderingen het construeren van een nieuwe lijst. De verenigingsoperatie maakt van twee lijsten één lijst. De operatie 'lid' wordt geïmplementeerd als een lineaire zoekactie in de lijst.

De implementatie met een bit-map maakt gebruik van een array van Booleaanse waarden, met één Boolean voor elk element van het totale domein. Als we ervan uitgaan dat tekens door acht bits worden voorgesteld, kan de tekenverzameling worden gerepresenteerd door een array van 256 Booleaanse waarden. De lege verzameling wordt voorgesteld door een array met alle elementen gelijk aan false. De operatie 'voeg_toe' zet het betreffende element van de array op true, en de operatie 'verwijder' zet het element op false. De verenigingsoperatie is het resultaat van de Booleaanse functie *of* op twee Booleaanse arrays.

Op deze twee klassieke implementaties van verzamelingen bestaan verschillende varianten. Eén van die varianten is het weergeven van het complement van de verzameling. De geketende lijst bevat dan de elementen die *niet* in de verzameling voorkomen; en de waarde true in de bit-map betekent dan, dat het element *niet* tot de verzameling behoort.

Een andere variant van de geketende lijst berust op het al of niet toelaten in de lijst van meer dan één exemplaar van dezelfde waarde.

De expressie

```
VOEG_TOE('X', VOEG_TOE('X', LEEG))
```

levert een verzameling met één element, maar bestaat de representatie uit een geketende lijst met één of met twee elementen? Als elementen hoogstens één keer in de geketende lijst voorkomen, dan is er nog keus tussen een geordende of een ongeordende lijst. Een geordende lijst wordt dan volgens een of ander criterium gesorteerd.

Elke representatie heeft zijn beperkingen, voordelen en nadelen. Representaties met een bit-map zijn alleen geschikt voor verzamelingen met een klein domein. De representatie met een geketende lijst is geschikt voor verzamelingen van elke grootte, behalve voor oneindige verzamelingen (met de complement-representatie kunnen alleen cofiniete verzamelingen binnen een oneindig domein worden voorgesteld). Overwegingen van tijd en ruimte verschillen belangrijk van implementatie tot implementatie. Aangezien het tekendomein eindig is, is elk van de bovenstaande representaties bruikbaar. In vele talen vormt het gegevenstype string een eenvoudige manier om verzamelingen tekens te representeren. Strings worden meestal efficiënter geïmplementeerd dan geketende lijsten van tekens en kunnen daarom goedkoper zijn. Om te laten zien hoe strings kunnen worden gebruikt om verzamelingen tekens te representeren, zullen we ervan uitgaan dat er een gegevensabstractie voor strings bestaat met concatenatie, substring-operaties en een operatie *lengte*. Merk op dat ons voorbeeld laat zien hoe gegevensabstracties kunnen worden gebruikt bij het opbouwen van andere gegevensabstracties. Gegevensabstracties kunnen op dezelfde manier als procedurele abstracties op een hiërarchische manier worden opgebouwd.

```
adt STRINGS is
     type STRING is private;

     LEGE_STRING: function return STRING;

     function "+" (A,B: STRING) return STRING;
     -- levert de concatenatie van A en B

     function "+" (A: STRING; B: CHAR) return STRING;
     -- levert de concatenatie van A en B

     function "+" (A: CHAR; B: STRING) return STRING;
     -- levert de concatenatie van A en B

     SUB: function (A: STRING; J: INTEGER) return CHAR;
     -- levert het Jde teken van string A
```

```
        SUBSTR: function (A: STRING; J,K: INTEGER) return STRING;
        -- levert de substring die met het Jde teken begint
        -- en met het Kde teken eindigt

        SUBSTR: function (A: STRING; J: INTEGER) return STRING;
        -- levert de substring die met het Jde teken begint
        -- en met het eind van de string eindigt

        LENGTE: function (A: STRING) return INTEGER;
        -- levert de lengte van string A

    end adt STRINGS;
```

De representatie van een string is gewoon een sequence van tekens en is daarom equivalent met de representatie van een geketende lijst. We moeten nu nog kiezen uit een aantal varianten, waarvan we er twee zullen laten zien: (1) ongeordende strings waarin tekens meermalen mogen voorkomen en (2) ongeordende strings waarin een teken maar hoogstens één keer mag voorkomen. De eerste variant is eenvoudiger te implementeren, aangezien we geen maatregelen behoeven te nemen tegen dubbele elementen:

```
    adt CHAR_SETS is
        private type CHARSET is STRING;

        LEGE_SET: function return CHARSET is
        begin
            return LEGE_STRING;
        end LEGE_SET;

        VOEG_TOE: function (C: CHAR; S: CHARSET) return CHARSET is
        begin
            return S+C;
        end VOEG_TOE;

        VERWIJDER: function (C: CHAR; S: CHARSET) return CHARSET is
            J: INTEGER;
            CH: CHAR;
            RESULTAAT: STRING := LEGE_STRING;
        begin
            for J:=1 to LENGTE(S) loop
                CH := SUB(S, J);
                if CH ≠ C then
                    RESULTAAT := RESULTAAT+C;
                end if;
            end loop;
            return RESULTAAT;
        end VERWIJDER;
```

```
        VERENIGING: function (SET1, SET2: CHARSET) return CHARSET is
        begin
            return SET1+SET2;
        end VOEG_TOE;

        LID: function (C: CHAR; S: CHARSET) return BOOLEAN is
            J: INTEGER;
        begin
            for J:=1 to LENGTE(S) loop
                if SUB(S, J)=C then
                    return TRUE;
                end if;
            end loop;
            return FALSE;
        end LID;

    end adt CHAR_SETS;
```

Merk op dat in de operatie VERWIJDER alle exemplaren van het teken moeten worden verwijderd, niet alleen het eerste. De operaties VERENIGING en VOEG_TOE zijn eenvoudig.

In de volgende implementatie (zonder dubbele elementen in de representatie) zijn de operaties VOEG_TOE en VERENIGING ingewikkelder en kosten ze meer tijd. In ruil daarvoor is de operatie VERWIJDER eenvoudiger en efficiënter. De operatie LID blijft onveranderd, maar wordt wel sneller doordat de te door-zoeken lijst even lang is of korter:

```
    adt CHAR_SETS is
        private type CHARSET is STRING;

        LEGE_SET: function return CHARSET is
        begin
            return LEGE_STRING;
        end LEGE_SET;

        VOEG_TOE: function (C: CHAR; S: CHARSET) return CHARSET is
        begin
            if LID(C, S) then
                return S;
            else
                return S+C;
            end if;
        end VOEG_TOE;

        VERWIJDER: function (C: CHAR; S: CHARSET) return CHARSET is
            J: INTEGER;
            CH: CHAR;
```

```
    begin
        for J:=1 to LENGTE(S) loop
            if SUB(S, J)=C then
                return SUBSTR(S,1,J-1)+SUBSTR(S,J+1);
            end if;
        end loop;
        return S;
    end VERWIJDER;

    VERENIGING: function (SET1, SET2: CHARSET) return CHARSET is
        J: INTEGER;
        RESULTAAT: CHARSET := SET1;
    begin
        for J:=1 to LENGTE(SET2) loop
            RESULTAAT := VOEG_TOE(SUB(SET2, J), RESULTAAT);
        end loop;
        return RESULTAAT;
    end VERENIGING;

    LID: function (C: CHAR; S: CHARSET) return BOOLEAN is
        J: INTEGER;
    begin
        for J:=1 to LENGTE(S) loop
            if SUB(S, J)=C then
                return TRUE;
            end if;
        end loop;
        return FALSE;
    end LID;

end adt CHAR_SETS;
```

In het ideale geval berust de keus van de implementatie uitsluitend op efficiëntie-overwegingen. De efficiëntie van beide bovenstaande implementaties hangt af van de frequentie waarmee de operaties worden gebruikt en ook van de eigenschappen van de gegevens. Daarom is de keus moeilijk analytisch te bepalen; die is misschien het best te maken op grond van een aantal tests.

Van de beide implementaties kan de ene de plaats van de andere innemen, doordat we met opzet hebben besloten gegevensabstracties te gebruiken. In Ada en andere talen met faciliteiten voor gegevensabstractie wordt deze onafhankelijkheid tussen gebruik en implementatie gegarandeerd met behulp van typecontrole. In andere talen moet dit op andere manieren worden afgedwongen.

Laten we, om te zien wat er gebeurt als deze onafhankelijkheid niet wordt afgedwongen, eens kijken naar een programmeur die de implementatie zonder dubbele elementen gebruikt en de volgende code schrijft:

```
S1, S2, S3, S4: CHARSET;
    ...
S3 := VERENIGING(S1, S2);
S4 := VERWIJDER('X', S3);
```

De programmeur ontdekt dat de twee toekenningsopdrachten in een lang durende herhalingslus staan. De programmeur meent ook dat in deze speciale situatie de verzamelingen S1 en S2 geen gemeenschappelijke elementen hebben. De programmeur herschrijft de code daarom als volgt:

```
S1, S2, S3, S4: CHARSET;
    ...
S3 := S1+S2;            -- snelle vereniging
S4 := VERWIJDER('X', S3);
```

Het ziet ernaar uit dat de programmeur alles correct heeft gedaan, maar de situatie is erg onveilig. Op de eerste plaats kan de programmeur ten onrechte denken dat S1 en S2 geen element gemeen hebben en dat ze misschien allebei een X bevatten. In dat geval wordt één van beide exemplaren van de X niet verwijderd en is de X na afloop nog steeds een element van S4! Bovendien blijven de beide implementaties die we tot nu toe hebben geschreven, misschien bruikbaar, maar er is geen garantie dat dat ook voor een andere implementatie zal gelden. Iemand zou kunnen besluiten dat een complement-implementatie beter is. Als die implementatie in dit geval wordt gebruikt, berekent de 'snelle vereniging' de doorsnee! Om deze redenen moet de bovenstaande code, hoewel die efficiënter is, worden afgeraden omdat deze versie op de lange duur moeilijker te onderhouden zal zijn. Zonder automatisch afgedwongen discipline of grote waakzaamheid van alle programmeurs kan een dergelijk ontwijken van een gegevensabstractie kostbaar worden.

Er bestaan nog vele andere implementaties van verzamelingen. Met andere gegevensstructuren, zoals gesorteerde binaire bomen, kan het sorteren van en zoeken in geordende lijsten efficiënter worden gemaakt. In paragraaf 4.4 hebben we even gekeken naar procedurele gegevenstypen. We sluiten deze paragraaf af met een andere representatie van verzamelingen die lijkt op procedurele gegevenstypen. Het is een algemene methode, die voor elk gegevenstype kan worden gebruikt. Deze ongebruikelijke implementatie heeft een aantal interessante eigenschappen. De operaties die een waarde van het betreffende type afleveren, doen niets ingewikkelds. De operanden worden opgeslagen en op een ander tijdstip uitgewerkt. Aangezien alleen LID geen CHARSET aflevert, is dat de enige

operatie die moet 'kijken hoe het gegeven eruit ziet'. Dat betekent dat de eerste vier operaties optimaal qua efficiëntie zijn en dat LID al het werk doet. Als LID niet wordt aangeroepen is de implementatie zeer snel, maar als LID vaak wordt aangeroepen kan hij erg langzaam worden.

Deze vorm van implementatie lijkt op *lazy evaluation* (luie evaluatie), voor het eerst genoemd door Burge (1975) en Friedman en Wise (1976). Bij lazy evaluation wordt van een expressie eerst de buitenste functie uitgewerkt in plaats van eerst de binnenste expressie. Argumenten van functies worden pas uitgewerkt als ze werkelijk nodig zijn. Onderstaande implementatie lijkt in zoverre op lazy evaluation dat de argumenten van het type CHARSET pas worden uitgewerkt als ze nodig zijn:

```
adt CHAR_SETS is
    type OP_TYPE is (LEGE_SET_OP, VOEG_TOE_OP, VERWIJDER_OP,
                    VERENIGING_OP);
    private type CHARSET is
        record
            case OP: OP_TYPE is
            when LEGE_SET_OP => ; -- geen velden
            when VOEG_TOE_OP =>  C: CHAR;
                                 S: pointer CHARSET;
            when VERWIJDER_OP => C: CHAR;
                                 S: pointer CHARSET;
            when VERENIGING_OP => SET1, SET2: pointer CHARSET;
            end case;
        end record;

    LEGE_SET: function return CHARSET is
        S: CHARSET;
    begin
        S.OP := LEGE_SET_OP;
        return S;
    end LEGE_SET;

    VOEG_TOE: function (C: CHAR; S: CHARSET) return CHARSET is
        SET: CHARSET;
    begin
        SET.OP := VOEG_TOE_OP;
        SET.C  := C;
        SET.S  := S;
        return SET;
    end VOEG_TOE;

    VERWIJDER: function (C: CHAR; S: CHARSET) return CHARSET is
        SET: CHARSET;
    begin
        SET.OP := VERWIJDER_OP
        SET.C  := C;
```

```
        SET.S  := S;
        return SET;
    end VERWIJDER;

    VERENIGING: function (SET1, SET2: CHARSET) return CHARSET is
        SET: CHARSET;
    begin
        SET.OP      := VERENIGING_OP;
        SET.SET1    := SET1;
        SET.SET2    := SET2;
        return SET;
    end VERENIGING;

    LID: function (C: CHAR; S: CHARSET) return BOOLEAN is
    begin
        case S.OP of
        when LEGE_SET_OP =>
            return FALSE;
        when VOEG_TOE_OP =>
            return C=S.C or else LID(C, S.S);
        when VERWIJDER_OP =>
            return C ≠ S.C and then LID(C, S.S);
        when VERENIGING_OP =>
            return LID(C, S.SET1) or else LID(C, S.SET2);
        end case;
    end LID;

end adt CHAR_SETS;
```

De gegevensstructuren die door de bovenstaande implementatie worden geconstrueerd worden door de andere operaties niet meer gewijzigd (dat wil zeggen dat de implementatie zich gedraagt als een éénmaal beschrijfbaar geheugen). Daardoor is de toekenning een waarde-toekenning, ook al worden er pointers gebruikt.

De manier van implementatie die hierboven is gebruikt, is algemeen en kan voor vrijwel alle gegevenstypen worden toegepast. Eerst wordt de verzameling bepaald van operaties die een waarde van het betreffende type afleveren. Op basis van die verzameling wordt een representatie opgebouwd die een union is waarin elk alternatief één van de operaties representeert. Elk alternatief heeft één veld voor elk operand van de operatie. Daarna wordt iedere operatie uitgeprogrammeerd doordat er een nieuwe waarde wordt opgebouwd en de operanden aan de juiste velden worden toegekend. Het moeilijkste deel van de methode is het ontwerpen en coderen van de andere operaties.

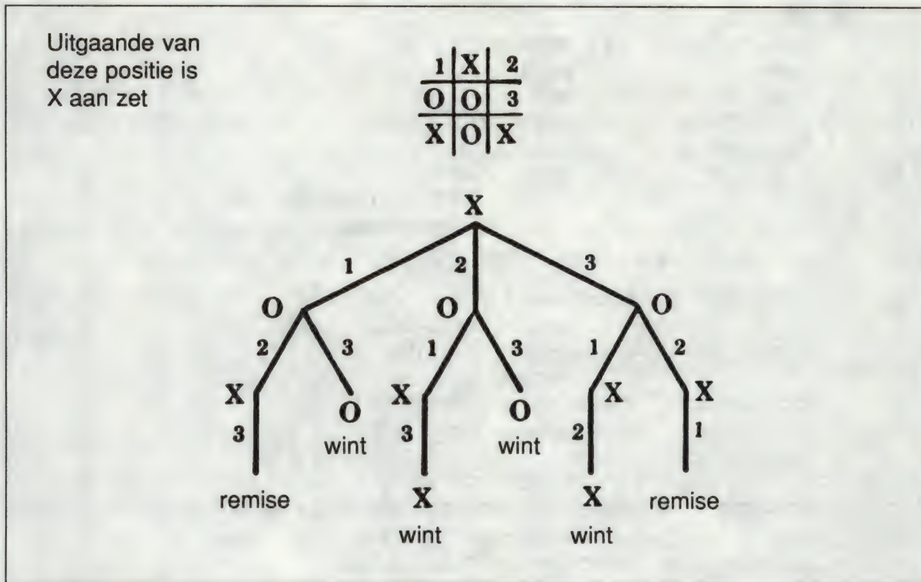
9.2 Spelletjes

Soms is het nuttig een groep van abstracte gegevenstypen te beschrijven in plaats van één bepaald gegevenstype. In deze paragraaf zullen we als voorbeeld een abstract gegevenstype *spel* definiëren. De semantiek van het abstracte gegevenstype wordt gevormd door de regels van het spel. Maar om een aantal algoritmen te beschrijven behoeven we alleen de syntaxis van het abstracte gegevenstype te kennen en slechts weinig van de semantiek. Daarom zullen we een groep van gegevenstypen ontwerpen die voor een grote klasse van spelletjes kan worden gebruikt. De semantiek van het gegevenstype wordt bepaald door de aard van een bepaald spel.

Spelletjes hebben vele gemeenschappelijke eigenschappen en een aantal algoritmen kunnen voor een groot aantal verschillende spelen gebruikt worden. Om deze overeenkomsten duidelijk te maken en de algemeen bruikbare algoritmen voor spelletjes te beschrijven, zullen we spelletjes organiseren als abstracte gegevenstypen. Bij het soort spelen dat we zullen beschouwen, houden we rekening met elk gewenst aantal spelers, met toevalsgebeurtenissen zoals het gooien met een dobbelsteen en met verborgen informatie. Verdere veronderstellingen over spelletjes zullen blijken als we het abstracte gegevenstype aan het ontwerpen zijn. Variaties op die veronderstellingen kunnen we verkrijgen door het ontwerp van het abstracte gegevenstype aan te passen.

De spel-abstractie bestaat uit vier typen. Het eerste type is een SITUATIE. Een situatie is een volledige beschrijving van een spel op een bepaald tijdstip. Die beschrijving bevat alle informatie die nodig is om verder te kunnen spelen. Bij boter-kaas-en-eieren bestaat die informatie uit de tekens die in elk van de negen hokjes zijn gezet en de speler die aan de beurt is. Het tweede en het derde type zijn ZET en SPELER. ZET geeft de mogelijke acties van de spelers weer. Het type SPELER is een opsomming van de spelers. We zullen aannemen dat in elke situatie er maar één speler aan de beurt kan zijn. Het resultaat van het doen van een zet is de overgang van de ene situatie naar de andere. Iedere situatie bepaalt een verzameling wettige zetten. Een speler voert zijn beurt uit door één van die wettige zetten te kiezen. Het spel moet beginnen en eindigen, dus zijn er begin- en eindsituaties. Het spel begint met een beginsituatie en eindigt met een eindsituatie. Vanuit een eindsituatie kan er geen zet worden gedaan.

De meeste spelletjes eindigen met winnaars en verliezers. Dat idee zal worden belichaamd in een type UITSLAG. Daarin kunnen winnaars en verliezers worden



Figuur 9-1 Een klein deel van de spelgraaf van boter-kaas-en-eieren.

aangegeven, of misschien nog andere soorten eindresultaat. Bij poker kan de uitslag bijvoorbeeld uit een geldbedrag bestaan.

Een spel kan grafisch worden weergegeven als een gerichte graaf waarin de punten de situaties representeren en de pijlen de zetten. Punten zonder binnenkomende pijlen zijn beginsituaties, en punten zonder uitgaande pijlen zijn eindsituaties. Bij elk punt kan worden aangegeven welke speler er aan de beurt is. Bij elk eindpunt (dat een eindsituatie representeert) kan in plaats daarvan de uitslag worden aangegeven. Figuur 9-1 laat een klein deel zien van de graaf voor boter-kaas-en-eieren. Toevalsgebeurtenissen kunnen op verschillende manieren worden weergegeven. Elke keer dat er een toevalsgebeurtenis vereist is, zou men een pseudo-speler *random* kunnen vragen een zet te doen. Of anders kan een zet niet slechts één overgang van situatie naar situatie representeren, maar vele van die overgangen. Wij zullen dit laatste alternatief gebruiken in ons ontwerp.

We kunnen de syntaxis van het abstracte gegevenstype SPEL nu volledig beschrijven als een waarde-gerichte implementatie. We gaan ervan uit dat er een gegevenstype LIST bestaat met operaties FIRST, REST en APPEND.

```
adt SPEL is
    type SITUATIE  is private;
    type ZET       is private;
    type SPELER    is private;
    type UITSLAG   is private;
    type ZET_LIJST is LIST(ZET);
    type SIT_LIJST is LIST( record
                              SIT: SITUATIE;
                              KANS: REAL;
                          end record );

    INIT_SIT       function
                       return SITUATIE;
    WIENS_ZET      function (SIT:SITUATIE)
                       return SPELER;
    GEN_ZETTEN     function (SIT:SITUATIE)
                       return ZET_LIJST;
    LEGALE_ZET     function (SIT:SITUATIE; Z: ZET)
                       return BOOLEAN;
    DOE_ZET        function (SIT:SITUATIE; Z: ZET)
                       return SIT_LIJST;
    BEKENDE_INFO   function (SIT:SITUATIE)
                       return SITUATIE;
    EINDSIT        function (SIT:SITUATIE)
                       return BOOLEAN;
    RESULTAAT      function (SIT:SITUATIE)
                       return UITSLAG;

end adt SPEL;
```

Een korte uitleg bij iedere functie is op zijn plaats.

1. INIT_SIT genereert een beginsituatie. Bij sommige spelen, zoals schaak en backgammon, produceert deze functie altijd dezelfde situatie. Voor andere spelen, zoals bridge, moet INIT_SIT een willekeurige beginsituatie genereren. Dat proces kan een simulatie nodig maken van het schudden en delen van een pak speelkaarten.

2. WIENS_ZET levert af welke speler in een bepaalde situatie als volgende aan de beurt is. In vele spelen voor twee spelers wisselt de beurt gewoon na elke zet. Maar dat gaat niet in het algemeen op. In het spelletje kamertje-verhuren krijgt een speler nog een beurt als hij of zij een hokje vol maakt. Onze procedure gaat ervan uit dat er in elke situatie maar één speler aan de beurt is.

3. GEN_ZETTEN genereert een lijst van alle mogelijke zetten die in een bepaalde situatie kunnen worden gedaan. Als er geen zetten zijn, wordt er een lege lijst afgeleverd. Dat kan alleen optreden in een eindsituatie. Onze procedure

gaat ervan uit dat er vanuit elke situatie maar een eindig aantal zetten kan worden gedaan. Die veronderstelling sluit spelletjes uit waarin een speler uit een oneindig groot aantal mogelijke zetten kan kiezen, zoals in kaartspelen met onbeperkt bieden. Als een speler een bod van elke gewenste grootte kan doen, komt dat neer op een keus uit een oneindig groot aantal mogelijkheden.

4. LEGALE_ZET gaat na of een zet geoorloofd is. De functie levert true af als de zet Z vanuit situatie SIT geoorloofd is. Deze procedure bevat dezelfde informatie als GEN_ZETTEN. De beide procedures moeten consistent zijn. Het is in feite mogelijk LEGALE_ZET te schrijven als:

```
LEGALE_ZET function (SIT:SITUATIE; Z: ZET) return BOOLEAN is
    LIJST_VAN_ZETTEN: ZET_LIJST := GEN_ZETTEN(SIT);
begin
    while LIJST_VAN_ZETTEN ≠ null loop
        if Z=FIRST(LIJST_VAN_ZETTEN) then
            return TRUE;
        end if;
        LIJST_VAN_ZETTEN := NEXT(LIJST_VAN_ZETTEN);
    end loop;
    return FALSE;
end LEGALE_ZET;
```

5. DOE_ZET voert een zet uit. De procedure berekent een nieuwe situatie op basis van de oude situatie en de zet. Als er geen toevalsgebeurtenissen zijn, wordt er een lijst afgeleverd van één situatie met een kans van 100%. Speelt het toeval wel een rol, dan wordt er een lijst afgeleverd waarin elke mogelijke situatie voorkomt met daarbij de kans dat die situatie optreedt. Het gooien met een dobbelsteen kan worden gerepresenteerd door het genereren van zes situaties met gelijke kansen. DOE_ZET voert alleen de opgegeven zet uit, maar bepaalt niet welke zet er wordt gespeeld, want dat is een strategische beslissing die de spelers maken.

6. BEKENDE_INFO levert de informatie af waar een speler recht op heeft. In een spel met volledige informatie, zoals schaak en backgammon, kan deze procedure de situatie gewoon onveranderd doorgeven. Maar in andere spelen, zoals bridge, moet BEKENDE_INFO geheime informatie, zoals de kaarten van de andere spelers, afdekken.

7. EINDSIT levert true af als de situatie een eindsituatie is, en anders false. Merk op dat deze functie, net als LEGALE_ZETTEN, in termen van GEN_ZETTEN kan worden uitgedrukt, omdat GEN_ZETTEN een lege lijst aflevert als de situatie een eindsituatie is.

8. RESULTAAT levert de uitslag van het spel af. Aan deze functie mogen alleen eindsituaties worden doorgegeven, anders wordt er een foutmelding gegeven. Voor de meeste spelen voor twee personen is de uitslag één van de mogelijkheden: (a) de eerste speler wint, (b) de tweede speler wint, of (c) remise.

Het abstracte gegevenstype SPEL is een formeel middel voor het beschrijven van de spelregels. Het beschrijft geen strategie en geeft niet aan hoe de spelers zouden kunnen spelen. Het beschrijft alle manieren waarop het spel zou kunnen verlopen.

Voor het beschrijven van een strategie hebben we een buiten het abstracte gegevenstype staande procedure nodig die, uitgaande van een situatie, een zet kiest. Elke speler zal een verschillende strategie hebben, dus laten we een array SPEEL van procedures nemen:

```
SPEEL: array (SPELER) of function (S: SITUATIE) return ZET;
```

Elke speler moet toegang hebben tot de openbare delen van het abstracte gegevenstype SPEL, maar niet tot de interne delen van het abstracte gegevenstype en ook niet tot de strategieën van de andere spelers.

We zijn nu in staat enkele algemene algoritmen met betrekking tot spelletjes te formuleren. De eerste algoritme is die van de onpartijdige scheidsrechter. De scheidsrechter coördineert het spel, vraagt de spelers hun zet te doen als ze aan de beurt zijn en geeft iedere speler de juiste informatie:

```
SCHEIDSRECHTER: function return UITSLAG is
     SIT: SITUATIE;
     VOLGENDE_ZET: ZET;
begin
     SIT := INIT_SIT();
     while not EINDSIT(SIT) loop
          VOLGENDE_ZET:=SPEEL(WIENS_ZET(SIT))(BEKENDE_INFO(SIT));
          if LEGALE_ZET(SIT, VOLGENDE_ZET) THEN
               SIT :=    -- maak random keus uit
                         -- DOE_ZET(SIT, VOLGENDE_ZET);
          else
               MELD_FOUT("Ongeoorloofde zet -- probeer nog eens");
          end if;
     end loop;
     return RESULTAAT(SIT);
end SCHEIDSRECHTER;
```

Een andere nuttige algoritme is de bekende minimax-procedure. Die wordt *minimax* genoemd omdat de ene speler probeert zijn of haar winst te maximaliseren, terwijl de andere speler probeert de winst van de eerste speler te minimaliseren. Het algoritme gaat ervan uit dat er geldt: (a) er zijn geen toevalsgebeurtenissen, (b) er zijn twee spelers, en (c) het is een spel met volledige informatie (dat wil zeggen een spel zonder verborgen informatie). Dus DOE_ZET levert één situatie af en BEKENDE_INFO is de identieke functie. De minimax-procedure kan worden gebruikt om de beste strategie voor elke speler te bepalen (zoals in de tweede procedure hieronder). Zoals de procedure hieronder is geïmplementeerd, levert hij de waarde van de uitslag van het spel af, ervan uitgaande dat beide spelers altijd optimaal spelen. De procedure

```
WAARDE: function (R: UITSLAG) return INTEGER;
```

levert een getal als resultaat dat de waarde van de uitslag aangeeft. Een groter getal is waardevoller voor de ene speler, een kleiner is waardevoller voor de andere speler. De getallen die bij een spel met drie mogelijke uitslagen worden gekozen, zouden er zo uit kunnen zien:

1. 100 als speler 1 wint;
2. 50 bij remise;
3. 0 als speler 2 wint.

De minimax-procedure wordt in zijn zuivere vorm (zoals hieronder) niet gebruikt, omdat hij bijna altijd te veel tijd kost. Beschouw een spel waarin er bij elke situatie *n* mogelijke zetten zijn en dat altijd *m* zetten duurt van beginsituatie tot eindsituatie. Dan wordt de minimax-procedure meer dan m^n keer aangeroepen!

```
MINIMAX: function (SIT: SITUATIE) return INTEGER is
     LIJST_VAN_ZETTEN: ZET_LIJST;
     VOLGENDE_ZET: ZET;
     V, W: INTEGER;
begin
     if EINDSIT(SIT) then
          return WAARDE(RESULTAAT(SIT));
     else
          LIJST_VAN_ZETTEN := GEN_ZETTEN(SIT);
          V := MINIMAX(DOE_ZET(SIT, FIRST(LIJST_VAN_ZETTEN)));
          while REST(LIJST_VAN_ZETTEN) ≠ null loop
               LIJST_VAN_ZETTEN := REST(LIJST_VAN_ZETTEN);
               VOLGENDE_ZET := FIRST(LIJST_VAN_ZETTEN);
               W := MINIMAX(DOE_ZET(SIT, VOLGENDE_ZET));
```

```
                if WIENS_ZET(SIT)=SPELER1 then
                    V := MAX(V, W);
                else
                    V := MIN(V, W);
                end if;
            end loop;
            return V;
        end if;
    end MINIMAX;

    BESTE_ZET: function (SIT: SITUATIE) return ZET is
        LIJST_VAN_ZETTEN: ZET_LIJST;
        V, W: INTEGER;
        BESTE, VOLGENDE_ZET: ZET;
    begin
        LIJST_VAN_ZETTEN := GEN_ZETTEN(SIT);
        V := MINIMAX(DOE_ZET(SIT, FIRST(LIJST_VAN_ZETTEN)));
        BESTE := V;
        while REST(LIJST_VAN_ZETTEN) ≠ null loop
            LIJST_VAN_ZETTEN := REST(LIJST_VAN_ZETTEN);
            VOLGENDE_ZET := FIRST(LIJST_VAN_ZETTEN);
            W := MINIMAX(DOE_ZET(SIT, VOLGENDE_ZET));
            if WIENS_ZET(SIT)=SPELER1 and W>V
            or WIENS_ZET(SIT)≠SPELER1 and W<V then
                V:=W;
                BESTE:=VOLGENDE_ZET;
            end if;
        end loop;
        return BESTE;
    end BESTE_ZET;
```

Omdat de minimax-procedure zo enorm veel tijd kost, is een snellere oplossing gewenst. De minimax-procedure levert een exact resultaat; als we tevreden zijn met een benadering kunnen we een snellere oplossing bereiken. En van de methoden is het invoeren van een functie SCHATTING die, gegeven een situatie, een benadering geeft van de waarde van de situatie — dat wil zeggen de procedure schat wat de minimax-functie als resultaat zou hebben afgeleverd. De functie SCHATTING hangt van het spel af; voor schaak kan de functie rekening houden met dingen als het aantal schaakstukken aan beide zijden en de positie aan beide zijden. Bij de minimax-procedure worden alleen de getallen als resultaat afgeleverd die door de functie WAARDE worden afgeleverd. Maar de functie SCHATTING kan elke waarde tussen de beide uitersten afleveren. Het ontwerpen en implementeren van zo'n procedure SCHATTING is moeilijk en moet van geval tot geval worden bekeken. De minimax-procedure en de schattingsprocedure worden zo gecombineerd, dat de minimax-procedure tot een bepaalde nestingsdiepte wordt gebruikt en dat daar dan de procedure SCHATTING wordt gebruikt.

De procedure MINIMAX wordt vervangen door de volgende functie.

```
BESTE_SCHATTING: function (NIVEAU: INTEGER; SIT: SITUATIE)
                              return INTEGER is
    LIJST_VAN_ZETTEN: ZET_LIJST;
    VOLGENDE_ZET: ZET;
    V, W: INTEGER
begin
    if NIVEAU=0 then
        return SCHATTING(SIT);
    end if;
    LIJST_VAN_ZETTEN := GEN_ZETTEN(SIT);
    V := BESTE_SCHATTING(NIVEAU-1,
                         DOE_ZET(SIT, FIRST(LIJST_VAN_ZETTEN)));
    while REST(LIJST_VAN_ZETTEN) ≠ null loop
        LIJST_VAN_ZETTEN := REST(LIJST_VAN_ZETTEN);
        VOLGENDE_ZET := FIRST(LIJST_VAN_ZETTEN);
        W := BESTE_SCHATTING(NIVEAU-1,
                             DOE_ZET(SIT, VOLGENDE_ZET));
        if WIENS_ZET(SIT)=SPELER1 then
            V := MAX(V, W);
        else
            V := MIN(V, W);
        end if;
    end loop;
    return V;
end BESTE_SCHATTING;
```

Naarmate de nestingsdiepte wordt vergroot zal de kwaliteit van het spel beter worden, maar de benodigde tijd zal exponentieel toenemen.

De bovenstaande verzameling functies laat zien hoe nuttig het is de details van het spel te scheiden van het abstracte idee van het spelen. We hebben een aantal verschillende algemeen toepasbare functies beschreven die voor een breed scala van spelen toepasbaar zijn. Elk spel kan onafhankelijk van de algemene functies worden geprogrammeerd. We besluiten deze paragraaf met één voorbeeld van een specifiek spelletje: boter-kaas-en-eieren. We zullen een situatie beschrijven als een lijst van tien elementen. Het eerste element geeft aan wie er aan de beurt is en de overige negen elementen geven de status van elk van de negen hokjes weer.

```
adt SPEL is

    type STATUS is (BLANCO, X, O);
    type SITUATIE is array (0..9) of STATUS;
    type ZET is 1..9;
    type SPELER is STATUS range X..O;
    type UITSLAG is (XWINT, OWINT, ONBESLIST);
```

```
INIT_SIT: function return SITUATIE is
    R: SITUATIE := (X,     BLANCO, BLANCO, BLANCO,
                           BLANCO, BLANCO, BLANCO,
                           BLANCO, BLANCO, BLANCO);

begin
    return R;
end INIT_SIT;

WIENS_ZET: function (SIT: SITUATIE) return SPELER is
begin
    return SIT(0);
end WIENS_ZET;

GEN_ZETTEN: function (SIT: SITUATIE) return ZET_LIJST is
    LIJST_VAN_ZETTEN: ZET_LIJST := LEGE_LIJST;
    J: INTEGER;
begin
    for J in 1..9 loop
        if SIT(J)=BLANCO then
            LIJST_VAN_ZETTEN := APPEND(LIJST_VAN_ZETTEN,J);
        end if;
    end loop;
    return LIJST_VAN_ZETTEN;
end GEN_ZETTEN;

LEGALE_ZET: function (SIT: SITUATIE; Z: ZET)
                     return BOOLEAN is
begin
    return SIT(Z)=BLANCO;
end LEGALE_ZET;

DOE_ZET: function (SIT: SITUATIE; Z: ZET) return ZET_LIJST is
     NIEUWE_SIT: SITUATIE;
begin
    if LEGALE_ZET(SIT, Z) then
        NIEUWE_SIT := SIT;
        NIEUWE_SIT(Z)  := WIENS_ZET(SIT);
        return MAAK_SIT_LIJST(NIEUWE_SIT, 100);
    else
        FOUT();
    end if;
end DOE_ZET;

BEKENDE_INFO: function (SIT: SITUATIE) return SITUATIE is
begin
    return SIT;
end BEKENDE_INFO;

EIND_SIT: function (SIT: SITUATIE) return BOOLEAN is
```

```
        begin
            -- lever true af als er geen blanco hokjes zijn
            -- of als er een rij van drie is;
        end EIND_SIT;

        RESULTAAT: function (SIT: SITUATIE) return UITSLAG is
        begin
            -- lever REMISE, X of O af op grond
            -- van de regels van boter-kaas-en-eieren;
        end RESULTAAT;

    end adt SPEL;
```

Opgaven

1. Voeg de gelijkheidsoperator '=' toe aan alle drie de implementaties van tekenverzamelingen in paragraaf 9.1.

2. Van welke veronderstelling over voorgedefinieerde operaties wordt er in de functie LEGALE_ZET uitgegaan? Wat is het gevolg als die veronderstelling onjuist is?

3. Schrijf een procedure waarmee elk spel kan worden gespeeld waarin de zetten van elke speler willekeurig worden gekozen. Maak geen gebruik van SPEEL of SCHEIDSRECHTER.

4. Beschouw het implementeren van een spel (zoals poker, nim, go, dammen, enzovoort) op basis van het abstracte gegevenstype SPEL. De algoritmen hangen meestal af van de representatie (dat wil zeggen de gegevensstructuren). Geef twee verschillende representaties en bespreek de verschillen in efficiëntie.

10

Polymorfisme

Polymorfisme, hetgeen veelvormigheid betekent, is een term die wordt gebruikt voor programmeertaalconstructies die op vele verschillende gegevenstypen betrekking hebben. Een polymorfe procedure bijvoorbeeld kan elk type parameter hebben en de aanroepen van de procedure kunnen waarden van verschillende typen doorgeven. Het type van de parameter wordt polymorf genoemd omdat de parameter waarden van verschillende typen kan hebben. Er zijn vele woorden die dezelfde betekenis hebben als polymorfisme. De talen CLU en Alphard gebruiken *type-generatoren* en Ada gebruikt de term *generiek. Geparametriseerde typen* en *schema's* zijn andere woorden die zijn voorgesteld. In dit hoofdstuk zullen we de termen 'polymorfisme' en 'polymorf' gebruiken.

In eerdere hoofdstukken hebben we vele voorgedefinieerde polymorfe typen gezien, die we type-constructors noemden. Zo is

```
array (N..M) of X
```

een polymorf type waarvan `N`, `M` en `X` de parameters zijn. Evenzo zijn de typen record, union en procedure polymorfe typen. De meeste talen hebben ook polymorfe operaties en procedures. Een polymorfe procedure heeft één of meer parameters die elk van vele verschillende typen kunnen zijn. Drie fundamentele polymorfe operaties zijn toekenning, array-indexering en functie-applicatie. Overloaded operatoren zijn geen voorbeelden van polymorfe functies. Het verschil zit in de implementatie. Bij een polymorfe functie behoort meestal maar

één algoritme dat de functie implementeert. Een overloaded functie heeft meestal voor elk type een verschillende algoritme. De operaties voor optelling en gelijkheid moeten bijvoorbeeld meestal voor elk type een verschillende algoritme gebruiken; er bestaat geen algemene algoritme voor alle soorten optelling of gelijkheid. Maar de operaties toekenning en functie-applicatie worden in het algemeen voor alle typen op één bepaalde manier geïmplementeerd.

In eerdere hoofdstukken hebben we door de gebruiker gedefinieerde typen geïntroduceerd. We breiden dat idee nu uit tot door de gebruiker gedefinieerde geparametriseerde typen. De technieken uit de vorige hoofdstukken zijn niet voldoende voor het creëren van constructor-typen zoals arrays. In het vorige hoofdstuk hebben we tekenverzamelingen ontworpen en geïmplementeerd, maar in dit hoofdstuk gaan we verder en bekijken we het implementeren van verzamelingen voor elk willekeurig gegevenstype.

10.1 Polymorfe parameters

Een polymorf type heeft één of meer parameters. Een polymorf type dat is voorzien van argumenten wordt een *instantie* van het polymorfe type genoemd, en zo'n instantie is een type van het polymorfe type. Het polymorfe type representeert dus een collectie typen en die collectie wordt gegenereerd door het scala van mogelijke waarden voor de parameters. Een polymorfe procedure heeft één of meer parameters van een polymorf type. Een polymorfe procedure representeert een collectie procedures, één voor elk type van de polymorfe typen. Om ervoor te zorgen dat polymorfe typen en procedures uitvoerbaar zijn, moet men de type-parameters opgeven om zo een niet-polymorf type of procedure te genereren. Die taak wordt *instantiatie* (of concretisering) genoemd, omdat zo een instantie van het polymorfe type of de polymorfe procedure wordt gecreëerd. Instantiatie vindt gewoonlijk plaats tijdens het compileren, zodat het nodig is dat de waarden van de parameters van de polymorfe typen en procedures al tijdens het compileren bekend zijn.

In de volgende voorbeelden zullen we identifiers in kleine letters gebruiken voor die waarden van type-parameters die gewoonlijk al tijdens het compileren bekend zijn. Later zullen we de syntactische aspecten en impliciete parameters bespreken. We zullen drie soorten parameters bekijken: waarden, typen en beperkte typen.

Waarden

In Pascal maken de grenzen van een array deel uit van het type. Aangezien Pascal eist dat alle typen tijdens het compileren bekend zijn, is het niet mogelijk een procedure te schrijven die als parameter een array accepteert van wisselende lengte. Als we ons een uitbreiding van Pascal voorstellen waarin procedures wel arrays van variabele lengte accepteren, dan zouden daarin ook de grenzen van zo'n array moeten worden meegegeven. De volgende procedure telt het aantal keren dat de waarde SLEUTEL in een array van willekeurige lengte voorkomt:

```
AANTAL_KEER: function (n, m: INTEGER;
                          X: array (n..m) of INTEGER;
                          SLEUTEL: INTEGER)
                    return INTEGER is
             TELLING: INTEGER := 0;
             J: INTEGER;
begin
             for J in n..m loop
                  if SLEUTEL=X(J) then
                       TELLING:=TELLING+1;
                  end if;
             end loop;
             return TELLING;
end AANTAL_KEER;
```

De parameters n en m zijn met opzet in kleine letters gespeld om te benadrukken dat ze bij het compileren als constanten bekend zijn. Bij elke aanroep van de procedure AANTAL_KEER is de waarde van de eerste twee parameters bekend, omdat in Pascal de grenzen van alle arrays bij het compileren vast staan; ze moeten bekend zijn om tijdens het compileren volledige typecontrole mogelijk te maken.

AANTAL_KEER is een eenvoudig voorbeeld van een polymorfe procedure. De derde parameter X representeert een collectie van mogelijke typen. Elke aanroep van deze procedure specificeert (via de eerste twee parameters) het type van de derde parameter.

Polymorfe typen kunnen numerieke parameters of andere value-parameters hebben die gebruikt worden voor het specificeren van zaken als lengte. Bekijk eens het volgende gegevenstype BAG, gerepresenteerd als een array van een zekere vaste lengte. De parameter max geeft het maximale aantal elementen aan dat in een bag kan worden opgeslagen.

```
type BAG (max: INTEGER) is
        record
                WAARDEN: array (1..max) of INTEGER;
                AANTAL: INTEGER;
        end record;

BAG1: BAG(100);  -- een voorbeeld van een declaratie met BAG
```

Typen

Een ander soort parameter is de type-parameter. Een bag van integers en een bag van tekens lijken in veel opzichten op elkaar. Het concept van een bag is onafhankelijk van het soort objecten dat erin wordt opgeslagen. Evenzo kan de procedure AANTAL_KEER gegeneraliseerd worden tot arrays van elk willekeurig type. Om deze generalisaties mogelijk te maken en ook alle parameters expliciet te maken moeten we een nieuw type *type* invoeren. Een korte bespreking daarvan is te vinden aan het eind van hoofdstuk 4. In dit hoofdstuk hebben we dit type nodig om alle type-parameters zichtbaar te maken. Beschouw als voorbeeld de volgende polymorfe procedure:

```
AANTAL_KEER: function (t: TYPE;
                       X: array (1..100) of t;
                       SLEUTEL: t)
                 return 0..100 is
        TELLING: 0..100 := 0;
        J: 1..100;
begin
        for J in 1..100 loop
                if SLEUTEL=X(J) then
                        TELLING:=TELLING+1;
                end if;
        end loop;
        return TELLING;
end AANTAL_KEER;
```

Deze procedure heeft drie parameters. De eerste geeft het type aan en de andere twee zijn van polymorfe typen die gebaseerd zijn op de eerste parameter. Net als de numerieke parameters uit de vorige paragraaf moeten de type-parameters ook als constanten bij het compileren vaststaan, zodat typecontrole tijdens het compileren mogelijk is. Maar als we naar de bovenstaande procedure kijken, zien we duidelijk dat als de argumenten van de juiste typen zijn, ook de instantiatie van AANTAL_KEER geen type-fouten bevat. Daaruit blijkt dat het niet nodig is dat de waarden van de type-parameters al tijdens het compileren bekend zijn. Hoe we elementen van type t met elkaar vergelijken zullen we later zien. Er kunnen typen zijn die geen vergelijkingsoperatie hebben.

Tot besluit van deze paragraaf geven we een voorbeeld van een geparametriseerd type BAG:

```
type BAG (t: TYPE) is
        record
                WAARDEN: array (1..100) of t;
                AANTAL: INTEGER;
        end record;

BAG1: BAG(INTEGER);  -- een voorbeeld van een declaratie met BAG
```

Typen met beperkingen op de operatoren

In de vorige paragraaf hebben we gemerkt dat er een moeilijkheid optreedt bij operaties die op een willekeurig type worden toegepast. Sommige polymorfe typen vereisen bepaalde operaties. We kunnen zulke eisen formaliseren door nog meer parameters toe te voegen. We kunnen de gelijkheidsoperator als parameter bij de procedure AANTAL_KEER opnemen, zodat er geen geheimzinnige operaties meer in de definitie van de procedure voorkomen.

```
AANTAL_KEER: function (t: TYPE;
                       X: array (1..100) of t;
                       SLEUTEL: t;
                       GELIJK: function (A, B: t)
                                     return BOOLEAN)
                return 0..100 is
        TELLING: 0..100 := 0;
        J: 1..100;
begin
        for J in 1..100 loop
                if GELIJK(SLEUTEL, X(J)) then
                        TELLING:=TELLING+1;
                end if;
        end loop;
        return TELLING;
end AANTAL_KEER;
```

In de parameterlijst van de procedure worden nu duidelijk alle eisen vermeld die aan het type t worden gesteld. Polymorfe typen vereisen vaak bepaalde operaties die op de elementen van de type-parameter moeten worden toegepast. Het is belangrijk dat die eisen samen met de type-parameters worden opgegeven en de meeste talen met polymorfisme beschikken over een speciale syntaxis voor dat doel.

Hier volgen voorbeelden uit enkele talen:

```
t: type where t has                                       -- CLU
       GELIJK: proctype(t, t) returns (bool);

t:form<GELIJK>                                            -- Alphard
type t with ( function GELIJK(t, t) : boolean;);          -- Russell
type t needs attributes                                   -- Schemes
       ( function GELIJK(t, t) returns boolean)

generic type t is private;                                -- Ada
       with function GELIJK(A, B: t)
           return BOOLEAN is GELIJK;
```

Enkele van deze constructies zijn ook anders te gebruiken. Zo kunnen die van Alphard en Russell bijvoorbeeld in elke declaratie gebruikt worden; ze geven dan een beperking aan op de klasse van operaties die kunnen worden gebruikt.

In de procedure AANTAL_KEER hebben we de functie GELIJK afzonderlijk van het type gespecificeerd. Voor geparametriseerde typen zouden we hetzelfde kunnen doen, maar in de meeste programmeertalen worden de operaties op een type bij het type aangegeven, zoals in bovenstaande voorbeelden. Deze aanpak is heel toepasselijk, omdat zulke operaties met recht kunnen worden beschouwd als deel van het type in plaats van als een tweede parameter. Maar de meeste van deze talen gaan niet ver genoeg. Alleen het syntactische gedeelte van de operator wordt gespecificeerd, maar vaak zijn ook bepaalde eigenschappen van de operatie noodzakelijk.

Een gelijkheidsoperatie heeft bepaalde eigenschappen die door het polymorfe type worden verondersteld. Het zou ideaal zijn als die eigenschappen volledig zouden moeten worden gespecificeerd. Alphard moedigt het specificeren aan van alle vereiste eigenschappen van elke operatie die bij een type-parameter behoort.

10.2 Kwesties in verband met typecontrole

In de vorige paragraaf hebben we verschillende soorten parameters bij polymorfe typen gezien. In deze paragraaf zullen we syntactische kwesties, semantische kwesties betreffende de macht van polymorfe typen en het soort typecontrole dat tijdens het compileren kan worden gedaan bekijken.

Statische en dynamische parameters

We moeten onderscheid maken tussen compile-time parameters en run-time parameters. Een *compile-time parameter* is een parameter waarvan de waarde altijd bij het compileren bekend is. De compiler gebruikt die informatie bij het compileren van het programma. Een *run-time parameter* is een parameter waarvan de waarde in het algemeen pas tijdens het uitvoeren van het programma bekend is, zodat de compiler code moet genereren voor het behandelen van alle mogelijke waarden. In de meeste talen zijn de typen statisch, dat wil zeggen dat in alle constructies elk type bij het compileren bekend is en alle type-parameters compile-time parameters zijn. Als alle typen statisch zijn, vereist dat dat de parameters van polymorfe typen compile-time waarden zijn. Dynamische typen zijn typen die tijdens de uitvoering van het programma worden gecreëerd en die de compiler in het algemeen niet kan voorspellen. Het toestaan van run-time parameters bij polymorfe typen zou ook dynamische typen mogelijk maken, omdat dan de waarde van de parameters pas tijdens het uitvoeren van het programma kan worden bepaald. Een taal met dynamische typen moet vele taken tot de uitvoering van het programma uitstellen die traditioneel door de compiler worden gedaan, zoals het vaststellen van de benodigde hoeveelheid geheugen, de typecontrole en de instantiatie van procedures. Wegens deze complicaties schrikken de meeste programmeertalen terug voor dynamische typen. Maar er zijn omstandigheden waaronder dynamische typen bijzonder nuttig zijn. Zo is het bijzonder nuttig als de grenzen van een array tijdens de uitvoering van het programma kunnen worden bepaald. Als de grenzen van een array tot het type behoren, is er een goede reden enkele dynamische typen toe te staan.

In de meeste gevallen zullen we aannemen dat parameters van polymorfe typen en procedures compile-time constanten zijn. Om onderscheid te maken tussen compile-time en run-time parameterwaarden nemen we de notatie over die gebruikt wordt door Tennent (1981). De parameters worden verdeeld in twee lijsten. De eerste lijst wordt tussen vierkante haken geplaatst en bevat alle compile-time parameters. De tweede lijst wordt tussen ronde haakjes geplaatst en bevat alle run-time parameters. Zo'n notatie helpt ons te zien wat er bij het compileren bekend moet zijn en welke waarden dynamisch zijn. De kop van de procedure AANTAL_KEER ziet er dan zo uit:

```
AANTAL_KEER: function [t: TYPE; n, m: INTEGER]
                      (X: array (n..m) of t; SLEUTEL: t)
              return INTEGER;
```

Een instantiatie van een polymorfe procedure of een polymorf type is nu gemakkelijker uit te leggen. Het komt er gewoon op neer dat de compile-time parameters bij de compilatie bekend zijn. Twee instanties van de procedure AANTAL_KEER zijn:

```
AANTAL_KEER[INTEGER, 1, 100]
```

en

```
AANTAL_KEER[CHARACTER, 5, 8]
```

Soms kan een instantiatie een nieuwe naam worden gegeven. Zo kan INT_AANTAL_KEER staan voor AANTAL_KEER[INTEGER, 1, 100]. Polymorfe typen zullen meestal alleen compile-time parameters hebben. Talen als SIMULA en CLU gebruiken een andere notatie, namelijk

```
INTEGER$AANTAL_KEER
```

in plaats van

```
AANTAL_KEER[INTEGER]
```

Expliciete en impliciete parameters

In de vorige paragrafen hebben we enkele verschillende syntactische methoden gezien voor het noteren van de parameters van polymorfisch typen en procedures. In deze paragraaf bekijken we nog een ander methode. Beschouw het volgende type:

```
type BAG[t: TYPE] is
          record
                    WAARDEN: array (1..100) of t;
                    AANTAL: 0..100;
          end record;

VOEG_TOE: procedure [t: TYPE] (B: BAG(t); X: t);
```

Op basis van dit polymorfe type kunnen we als volgt twee instanties van het type BAG creëren:

```
type INT_BAG is BAG[INTEGER];
type CHAR_BAG is BAG[CHARACTER];
```

```
BAG1: INT_BAG;
BAG2: CHAR_BAG;
```

De procedure VOEG_TOE moet met een type geïnstantieerd worden. We kunnen dat expliciet doen door het type van het eerste argument door te geven. We hebben daarvoor al verschillende syntactische methoden genoemd:

```
VOEG_TOE(INTEGER, BAG1, 5);
VOEG_TOE[INTEGER] (BAG1, 5);
INTEGER$VOEG_TOE(BAG1, 5);
```

Een impliciete aanpak verloopt als volgt. Het is heel natuurlijk om de procedure VOEG_TOE zo te gebruiken:

```
VOEG_TOE(BAG1, 5);
VOEG_TOE(BAG2, 'A');
```

Elke procedure-aanroep bevat voldoende informatie om te bepalen welke procedure VOEG_TOE er bedoeld wordt. De compiler kan meestal zelf de ontbrekende parameters aanvullen. De compiler kan de naam VOEG_TOE opvatten als een overloaded operator en dan met behulp van de algoritme voor operator-identificatie vaststellen om welke procedure het gaat.

De semantiek van polymorfisme

Polymorfisme wordt vaak gezien als een syntactische aangelegenheid. Polymorfe procedures en typen worden dan beschouwd als macro's met parameters die tijdens het compileren worden geëxpandeerd. Na het expanderen van alle polymorfe procedures en typen verricht de compiler de normale parsing en type-analyse. In dit licht bezien heeft polymorfisme geen semantische betekenis, maar laten we eens kijken naar de semantische implicaties van deze syntaxisgerichte opvatting. Voor ieder polymorf type moet worden bepaald welke instantiaties er door het programma worden vereist. Als het niet de verantwoordelijkheid van de programmeur is om die instanties aan te geven, zoals dat in Ada wordt verlangd, dan moet de compiler die taak uitvoeren. Nadat de compiler een lijst van instanties heeft opgesteld, kopieert en instantieert de compiler alle polymorfe routines en typen met de juiste statische waarden voor de parameters. Die taak kan moeilijk zijn als er impliciete parameters worden gebruikt. Na deze fase zijn alle polymorfe typen en procedures vervangen door collecties niet-polymorfe (dat wil zeggen normale) typen en procedures. Al deze afgeleide typen en procedures worden gecompileerd alsof ze op zichzelf staan. Be-

paalde moeilijkheden die met macro-preprocessors optreden, treden tijdens het boven beschreven proces ook op. Recursieve polymorfe typen en procedures kunnen niet worden geëxpandeerd, net als recursieve macro's niet geëxpandeerd kunnen worden. Sommige instanties van een polymorf type kunnen geoorloofd zijn en andere niet. Er kan bijvoorbeeld een type-fout zijn die alleen ontdekt wordt als er bepaalde type-parameters worden gebruikt. Een ander nadeel van een syntactisch gezichtspunt is dat er tijdens het compileren en tijdens het uitvoeren van het programma veel dubbel gebeurt, aangezien alle afgeleide typen en procedures onafhankelijk worden behandeld. Er worden vrijwel identieke functies gecompileerd en voor elke instantie van een polymorf type of een polymorfe procedure moet een bijna identieke code worden opgenomen. Door handige optimalisatie kan iets van dat dubbele werk worden teruggedraaid, maar duplicatie is een onvermijdelijk gevolg van het syntactische standpunt.

Een semantisch standpunt ten opzichte van polyformisme ondervangt vele van de bovenstaande nadelen. Onder een semantische opvatting van polymorfisme kan de typecontrole van polymorfe typen en procedures gebeuren voordat ze geïnstantieerd worden. Bij een semantische opvatting zijn efficiënte implementaties mogelijk, omdat de polymorfe typen niet behoeven te worden gedupliceerd. Recursief polymorfisme wordt ook mogelijk.

Onder een semantisch gezichtspunt worden polymorfe typen en procedures geparseerd als een deel van de taal (in plaats van behandeld te worden als een syntactisch suikerlaagje op een niet-polymorfe taal). Bij het compileren van polymorfe procedures behoort meestal ook typecontrole van de definitie van de procedure voordat de instantiatie plaatsvindt. Ook bij elk gebruik van een polymorfe procedure moet typecontrole plaatsvinden, maar, in tegenstelling tot de situatie bij het syntactische gezichtspunt, vereist de aanroep van een polymorfe procedure nu niet dat we voor elke instantiatie typecontrole uitvoeren op de definitie van de procedure. Op die manier kan de benodigde hoeveelheid typecontrole aanzienlijk worden verminderd.

Een implementatie van polymorfe procedures op een semantische basis is flexibeler en kan tot aanzienlijke optimalisaties leiden. En mogelijkheid is het syntactische uitgangspunt te imiteren en de code bij iedere instantiatie te dupliceren. Een andere mogelijkheid is van de statische parameter een run-time parameter te maken. Die extra parameter kan in de code worden gebruikt om het juiste stuk code te selecteren, dat van instantie tot instantie kan verschillen. De optimale implementatie kan bestaan uit een combinatie van deze beide uitersten.

Ter illustratie van enkele van deze semantische overwegingen zullen we eens kijken naar de volgende polymorfe procedure:

```
AANTAL_KEER: function [t: TYPE]
                    (X: array (1..100) of t; SLEUTEL: t)
                return 0..100 is
        TELLING: 0..100 := 0;
        J := 1..100;
begin
        for J in 1..100 loop
                if SLEUTEL=X(J) then
                        TELLING:=TELLING+1;
                end if;
        end loop;
        return TELLING;
end AANTAL_KEER;
```

Zonder zelfs maar te kijken hoe AANTAL_KEER wordt gebruikt kan de compiler een typecontrole uitvoeren op de definitie van de procedure. Op grond van de definitie van de procedure kan de compiler vaststellen dat de enige eis die aan t wordt gesteld, is dat er een operator = bij hoort die twee operanden van type t heeft en een waarde van het type BOOLEAN aflevert. Beschouw de volgende twee aanroepen van de functie AANTAL_KEER:

```
LEEFTIJDEN:   array (1..100) of INTEGER;
NAMEN:        array (1..100) of STRING;

begin
        ...
        AANTAL_KEER(LEEFTIJDEN, 5);
        AANTAL_KEER(NAMEN, "Alfa");
```

Op grond van de declaraties van LEEFTIJDEN en NAMEN (en op grond van de letterlijke constanten) kan de compiler vaststellen dat de polymorfe procedure AANTAL_KEER de type-parameters INTEGER respectievelijk STRING heeft. Beide typen voldoen aan de eisen die aan de type-parameter t worden gesteld, dus zijn alle typen nu correct.

We kunnen nu eens kijken naar de code die zou kunnen worden gegenereerd door een compiler die dit speciale voorbeeld implementeert. De methode met verdubbeling van de code leidt tot het kopiëren van de procedure, éénmaal voor elk type dat wordt gebruikt. Dat zou ook gebeuren als we te werk zouden gaan vanuit het syntactische gezichtspunt. Voor bovenstaand voorbeeld zouden er twee kopieën van AANTAL_KEER worden gegenereerd, met als enige verschil in de code het adres van de gelijkheidsroutine. In de ene kopie zou de gelijkheids-

routine voor strings worden gebruikt, in de andere de gelijkheidsroutine voor integers. Elke kopie van de routine zou een unieke naam (of adres) krijgen en een aanroep van AANTAL_KEER zou code genereren die één van beide kopieën aanroept. Deze implementatie kan er als volgt uitzien:

```
AANTAL_KEER1: function (X: array (1..100) of INTEGER;
                        SLEUTEL: INTEGER)
        ...

AANTAL_KEER2: function (X: array (1..100) of STRING;
                        SLEUTEL: STRING)
        ...

AANTAL_KEER1(LEEFTIJDEN, 5);
AANTAL_KEER2(NAMEN, "Alfa");
```

Een andere aanpak is de volgende. De compiler genereert code die van de typeparameter in feite een extra run-time parameter maakt. We zullen dat de aanpak met *gedeelde* code noemen. Deze implementatie voegt de code aan elkaar en zal zo minder geheugen vergen, maar meer tijd kosten voor het behandelen van de extra parameter van AANTAL_KEER. Het eindresultaat van de aanpak met gedeelde code komt neer op een procedure met een extra, door de compiler toegevoegde run-time parameter die het type specificeert. Het volgende is een illustratie van deze aanpak.

```
AANTAL_KEER: function (HET_TYPE: INTEGER;
                       X: array (1..100) of IETS;
                       SLEUTEL: IETS)
                return 0..100 is
         TELLING: 0..100 := 0;
         J := 1..100;
         GELIJK: procedure (X, Y: IETS) return BOOLEAN is
         begin
                if HET_TYPE=1 then
                        INT_GELIJKHEID(X, Y);
                else
                        STRING_GELIJKHEID(X, Y);
                end if;
         end GELIJK;

begin                  -- definitie van functie AANTAL_KEER
         for J in 1..100 loop
                if GELIJK(SLEUTEL, X(J)) then
                        TELLING:=TELLING+1;
                end if;
         end loop;
         return TELLING;
end AANTAL_KEER;
```

```
...
AANTAL_KEER(1, LEEFTIJDEN, 5);
AANTAL_KEER(2, NAMEN, "Alfa");
```

De keuze van de implementatie hangt af van de programmeertaal, de compiler, de programmeur en/of de speciale eisen van het programma. Een compiler zal misschien altijd de aanpak met gedeelde code kiezen, maar een optimaliserende compiler zou tot een beter resultaat kunnen komen. Als een polymorfe routine maar met één enkele waarde voor een bepaalde type-parameter wordt gebruikt, dan is het duidelijk dat dat type moet worden geïnstantieerd. Voor recursieve polymorfe routines is er geen keus (tenzij er een maximale diepte bekend is): die moeten met gedeelde code worden geïmplementeerd. Als een polymorfe routine met vele verschillende type-parameters wordt gebruikt en elke versie maar een paar keer wordt aangeroepen tijdens de uitvoering van het programma, dan is de aanpak met gedeelde code de beste. Als één van die versies zeer vaak wordt gebruikt, dan is het het best dat voor die ene waarde van de type-parameter instantiatie wordt uitgevoerd en dat voor alle andere gevallen de methode met gedeelde code wordt gebruikt. Het is in het algemeen niet duidelijk hoe we automatisch kunnen vaststellen welke implementatie-strategie voor een willekeurig programma het beste is.

Het uitwerken van de details van een semantische opzet voor polymorfisme is niet altijd gemakkelijk. Als men bereid is bepaalde beperkingen op het polymorfisme te aanvaarden, kan men de semantiek van polymorfisme op een vrij gemakkelijke manier definiëren. Maar wil men gegeneraliseerd polyformisme bereiken, waarin ook recursiviteit is toegestaan, dan komen daarbij vele wiskundige details te pas. Deze details worden soms bestudeerd in een eenvoudiger context, zoals de lambda-calculus.

Type-inferentie

Een veel toegepaste stijl van polymorfisme, ontwikkeld door Robin Milner (1978) en gebruikt in talen als ML en HOPE, kan worden gekarakteriseerd als polymorfisme met sterke typecontrole, maar zonder declaraties. In plaats van variabelen met een bepaald type te declareren, gebruikt de programmeur de variabelen gewoon en leidt de compiler het type van de variabelen af. Gegeven een willekeurige expressie met variabelen van onbekend type, kent een *type-toekenning* een type toe aan elke variabele. Een type-toekenning is geldig als deze leidt tot correct getypeerde expressies. *Type-deductie* is het bepalen van

alle mogelijke geldige toekenningen. Bij de expressie X=5 kan men bijvoorbeeld afleiden dat het type van X integer moet zijn, omdat alleen integers vergeleken kunnen worden met integers. Maar voor de variabelen in de expressie X=Y bestaan er vele geldige type-toekenningen. Als we aannemen dat de operanden van de gelijkheidsoperator hetzelfde type moeten hebben, kunnen we afleiden dat het type van X en van Y gelijk moet zijn. In een type-schema kan deze voorwaarde worden vastgelegd. Een *type-variabele* stelt een type voor en expressies met type-variabelen stellen verzamelingen van typen voor. Voor type-variabelen zullen we Griekse letters gebruiken. Een *type-schema* kent aan elke variabele een type-expressie toe. Een instantie van een type-schema bestaat uit het systematisch vervangen van elke type-variabele door een bepaald type. Als elke instantie van een type-schema tot een geldige type-toekenning leidt, dan is ook het type-schema geldig. Voor het meest algemene geldige type-schema, dat de *hoofdtypering* of het *hoofdschema* wordt genoemd, geldt dat elke geldige type-toekenning een instantie van het type-schema is. Het hoofdschema voor bovenstaand voorbeeld is

X: α
Y: α

waarin α een type-variabele is. Dit schema legt de beperking vast dat het type van X gelijk moet zijn aan het type van Y. Beschouw nu de ingewikkelder expressie

```
F(A, G(C)) = G(A)
```

De hoofdtypering is

A: α
C: α
F: $\alpha \times \beta \rightarrow \beta$
G: $\alpha \rightarrow \beta$

waarin $\alpha \times \beta \rightarrow \gamma$ slaat op een functie met twee parameters van de typen α en β die een waarde van het type γ aflevert. Zolang de type-variabelen α en β systematisch worden vervangen door een bepaald type blijft de expressie correct getypeerd. Elke andere type-toekenning leidt tot incorrect getypeerde expressies.

Hindley (1969) gebruikte de unificatie-algoritme om automatisch het hoofdschema te deduceren. Deze methode kan worden gebruikt als basis voor het invoeren van polymorfisme in een programmeertaal. Als voor het vastleggen van

het type van een expressie type-variabelen worden gebruikt, betekent dat dat de expressie polymorf is. Type-variabelen zijn geen variabelen van de programmeertaal en de programmeertaal kent ook geen type *type* met typen als waarden; de type-variabelen worden uitsluitend gebruikt om polymorfisme in te voeren door het vastleggen van klassen van typen. Zo'n expressie kan pas worden geëvalueerd als van alle type-variabelen de waarde is bepaald; dat treedt van nature op als het programma wordt gecompileerd of uitgevoerd. Expressies kunnen polymorf zijn, maar waarden niet.

In ML kunnen in een polymorfe procedure parameters worden gedeclareerd met of zonder type-variabelen. Beschouw het voorbeeld met AANTAL_KEER waarin de arrays zijn vervangen door functies. Deze functie kan in de stijl van ML herschreven worden als

```
AANTAL_KEER: function (X: function (Y: INTEGER) return α;
                       SLEUTEL: α )
                  return INTEGER is
     ...
end AANTAL_KEER;
```

of als

```
AANTAL_KEER: function (X; SLEUTEL) is
begin
        TELLING := 0;
        for J in 1..100 loop
                if SLEUTEL=X(J) then
                        TELLING:=TELLING+1;
                end if;
        end loop;
        return TELLING;
end AANTAL_KEER;
```

In beide bovenstaande voorbeelden stelt de compiler vast dat AANTAL_KEER een polymorfe functie is van het type

```
( INTEGER  →  α ) × α  → INTEGER
```

In het tweede voorbeeld geldt dit omdat SLEUTEL hetzelfde type moet hebben als X(J) en omdat de functie TELLING als resultaat aflevert. Het type van een variabele kan vaak worden afgeleid uit de manier waarop de variabele wordt gebruikt. ML volgt de filosofie dat declaraties niet verplicht moeten zijn in gevallen waarin het type uit de context kan worden afgeleid. Deze conventie leidt tot compactere programma's en leidt bovendien tot een natuurlijke manier om po-

lymorfisme in te voeren zonder dat daarvoor meer nieuwe syntaxis nodig is dan type-variabelen.

ML gaat uit van een semantische opvatting van polymorfisme in plaats van een syntactische opvatting (alleen op macro-achtige expansie gebaseerd). Zo wordt in ML bij polymorfe procedures de typecontrole uitgevoerd vóór de instantiatie en zijn er ook recursieve polymorfe typen en procedures toegestaan. Maar tijdens het uitvoeren van het programma zijn er geen polymorfe procedures of waarden. Voor de evaluatie van een expressie worden alle type-variabelen door concrete typen vervangen. Dat maakt het gebruik van polymorfe procedures binnen andere polymorfe procedures niet onmogelijk; het is alleen vereist dat bij het evalueren van een expressie er geen type-variabelen meer zijn waarvan de waarde nog niet is vastgesteld.

Een nadeel van Milners aanpak van typecontrole is dat polymorfisme daarin uitsluitend is gebaseerd op type-parameters. De grenzen en de omvang van arrays kunnen niet als type-parameters worden gebruikt. Bovendien zijn er geen type-parameters mogelijk die operaties meevoeren. In de voorbeelden in de volgende paragraaf zien we de gevolgen daarvan. Leivant (1983a) heeft het idee van type-deductie verder ontwikkeld met zaken als type-coërcies en overloading.

10.3 Voorbeeld van polymorfe gesorteerde lijsten

Om het gebruik van polymorfe typen in een aantal verschillende talen te illustreren volgt hier een eenvoudig voorbeeld. De gegevensabstractie *gesorteerde lijst* is een geordende verzameling waarden. Twee belangrijke parameters zijn het type van de te sorteren waarden en de ordening. Integers bijvoorbeeld kunnen in stijgende of in dalende volgorde worden gezet en strings kunnen naar lengte of in lexicografische volgorde worden gesorteerd. We zullen drie operaties op gesorteerde lijsten bekijken.

lege_lijst()	creëert een lege gesorteerde lijst;
voeg_toe(w, gl)	voegt een nieuwe waarde w toe aan de gesorteerde lijst gl
doorloop(gl, p)	past procedure p toe op elke waarde in de gesorteerde lijst gl

Het polymorfe type gesorteerde lijst heeft twee parameters: het type van de te sorteren waarden, dat we WTYPE zullen noemen, en de ordeningsoperatie, die

twee parameters van het type WTYPE heeft en een Booleaanse waarde als resultaat levert. Laat het symbool '≤' de ordeningsoperatie representeren. Die operatie moet de eigenschappen van een totale ordeningsrelatie hebben, namelijk voor alle *a, b* en *c* moet gelden:

$a \leq b$ en $b \leq c$ impliceert $a \leq c$
$a \leq b$ en $b \leq a$ impliceert $a = b$
$a \leq b$ of $b \leq a$

De ordeningsoperatie is een parameter van het abstracte gegevenstype gesorteerde lijst. Voor een goede werking moet de ordeningsoperatie deze eigenschappen hebben, maar er bestaat tegenwoordig nog geen praktisch toepasbaar systeem dat deze eisen automatisch kan afdwingen. In recente werkzaamheden aan Clear (door Burstall en Goguen, 1977), aan Larch Shared Language (door Guttag en Horning, 1983) en aan OBJ2 (door Goguen, 1984) worden veelbelovende systemen verkend die zulke eisen wel afdwingen.

Een belangrijk punt dat we in onze voorbeelden zullen tegenkomen is hoe en waar we de ordeningsparameter moeten specificeren. Als de taal geparametriseerde gegevensabstracties kent, ligt die manier misschien voor de hand. In talen zonder geparametriseerde typen kan de ordeningsrelatie als parameter worden doorgegeven aan één van de drie operaties. Op het eerste gezicht is *voeg_toe* de enige operatie die de ordeningsoperatie nodig heeft en daarom lijkt het er misschien op dat we de ordeningsoperatie als derde parameter aan *voeg_toe* moeten meegeven. Maar dat blijkt de slechtste plek te zijn om de ordeningsoperatie door te geven. Dat komt doordat er geen garantie bestaat dat bij elke aanroep van *voeg_toe* dezelfde ordeningsoperatie wordt meegegeven. Als er verschillende ordeningsoperaties worden meegegeven, onstaat er een gemengde lijst. Om de gesorteerde lijst robuuster te maken is het beter de ordeningsrelatie mee te geven aan de procedure *lege_lijst* of aan *doorloop*. Als we de ordeningsoperatie doorgeven aan *doorloop*, betekent dat dat we de gegevens niet gesorteerd opslaan. In plaats daarvan sorteren we de lijst elke keer dat we de gegevens doorlopen. Een voordeel daarvan is dat we dezelfde lijst volgens verschillende ordeningen kunnen doorlopen. Als we de ordeningsoperatie aan *lege_lijst* doorgeven, houdt dat in dat we de ordening pas bij de volgende lege lijst kunnen veranderen. In ruil daarvoor zijn we in staat efficiëntere sorteermethoden te implementeren, en hoeft *doorloop* de gegevens niet bij elke aanroep te sorteren. In onderstaande implementaties zullen we de ordeningsoperatie, in gevallen waarin die geen parameter van de gegevensabstractie kan zijn, doorgeven aan de procedure *lege_lijst*.

Gesorteerde lijsten in de stijl van PL/I of C

Onze eerste implementatie van de gesorteerde lijst is geschreven in een taal die op C of PL/I lijkt en ongetypeerde pointers heeft. Deze talen hebben geen polymorfe typen, maar we kunnen een goede gelijkenis met polymorfisme bereiken door ongetypeerde pointers te gebruiken om willekeurige typen mee te representeren en door de ordeningsrelatie door te geven aan de procedure *lege_lijst*. Net als in eerdere voorbeelden zullen we de naam IETS gebruiken om ongetypeerde pointers aan te duiden. Deze implementatie eist dat alle waarden in een gesorteerde lijst pointers naar waarden zijn. Verder zullen we uitgaan van een variabele-gerichte abstractie, aangezien dat overeenkomt met de programmeerstijl in deze talen.

```
adt GESORTEERDE_LIJSTEN is

    type ORDENING is function (X, Y: IETS) return Boolean;

    type GESORTEERDE_LIJST is
        record
            VOLGORDE: ORDENING;
            WAARDEN: LIJST_VAN_IETS;
        end record;

    type LIJST_VAN_IETS is pointer
        record
            EERSTE: IETS;
            REST: LIJST_VAN_IETS;
        end record;

    LEGE_LIJST: procedure (GL: out GESORTEERDE_LIJST;
                           VOLGORDE_OPERATIE: ORDENING) is
    begin
        GL.VOLGORDE := VOLGORDE_OPERATIE;
        GL.WAARDEN := null;
    end LEGE_LIJST;

    VOEG_TOE: procedure (W: IETS; GL: in out GESORTEERDE_LIJST) is
    T, VORIGE: LIJST_VAN_IETS;
    begin
        if GL.WAARDEN = null then
            GL.WAARDEN:= new LIJST_VAN_IETS(W, null);
        else
            T:= GL.WAARDEN;
            while T ≠ null loop
                if GL.VOLGORDE(W, T.EERSTE) then
                    T.REST := new LIJST_VAN_IETS(T.EERSTE,T.REST);
                    T.EERSTE := W;
                    return;
                end if;
```

```
                    VORIGE := T;
                    T:= T.REST;
               end loop;
               VORIGE.REST := new LIJST_VAN_IETS(W, null);
          end if;
     end VOEG_TOE;

     DOORLOOP: procedure (GL: GESORTEERDE_LIJST;
                          PROC: procedure (X: IETS)) is
          T: LIJST_VAN_IETS;
     begin
          T:= GL.WAARDEN;
          while T ≠ null loop
               PROC(T.EERSTE);
               T := T.REST;
          end loop;
          end DOORLOOP;

end adt GESORTEERDE_LIJSTEN;
```

In dit voorbeeld is het type IETS een pointer naar een willekeurige waarde. Tot de operaties op waarden van het type IETS behoort de gelijkheidoperatie (eigenlijk alleen gelijkheid met null) en pointer-toekenning. De enige operatie op de waarden waarnaar IETS wijst is ORDENING.

Merk op dat dit geen macro-implementatie is, zodat van elke procedure maar één exemplaar wordt gecompileerd, niet één kopie voor elke soort gesorteerde lijst. Verder voert de compiler geen typecontrole uit en wordt er ook tijdens het draaien van het programma geen typecontrole verricht. Zo bevat het volgende programmasegment, dat de procedures voor gesorteerde lijsten gebruikt, een typefout die door de meeste PL/I-compilers niet wordt ontdekt:

```
type INT_KNOOP is ...             -- een element van een
                                  -- gesorteerde lijst van integers
type STR_KNOOP is ...             -- een element van een
                                  -- gesorteerde lijst van strings

INT_ELEM: INT_KNOOP;              -- een integer knoop

INT_LIJST: GESORTEERDE_LIJST;  -- een gesorteerde lijst van - integers
STR_LIJST: GESORTEERDE_LIJST;  -- een gesorteerde lijst van - strings

VOEG_TOE(INT_ELEM, STR_LIJST); -- een typefout die door PL/I
                               -- niet wordt ontdekt
```

De stijl van Ada

We zullen nu eens kijken naar talen met polymorfe gegevensabstracties en een syntactisch uitgangspunt ten aanzien van polymorfisme. Ada is een goed voorbeeld van zo'n taal, waarin expliciete instantiatie van polymorfe typen en procedures vereist is. In tegenstelling tot PL/I kent Ada volledige typecontrole en mag de ordeningsoperatie een parameter van de gegevensabstractie zijn in plaats van een operatie. Eveneens in tegenstelling tot PL/I hoeven de waarden in Ada geen pointers te zijn. In de geest van Ada zou een gesorteerde lijst er zo uit kunnen zien:

```
generic type WTYPE is private;
    with function "<" (X, Y: WTYPE) return BOOLEAN is "<";
package GESORTEERDE_LIJSTEN is

    type GESORTEERDE_LIJST is pointer
        record
            EERSTE: WTYPE;
            REST: GESORTEERDE_LIJST;
        end record;

    LEGE_LIJST: procedure (GL: out GESORTEERDE_LIJST) is
    begin
        GL := null;
    end LEGE_LIJST;

    VOEG_TOE: procedure (W: WTYPE;
                         GL: in out GESORTEERDE_LIJST) is
        T: GESORTEERDE_LIJST;
    begin
        ...
    end VOEG_TOE;

    DOORLOOP: procedure (GL: GESORTEERDE_LIJST;
                         PROC: procedure (X: WTYPE)) is
        ...
    end DOORLOOP;

end package GESORTEERDE_LIJSTEN;
```

Het gebruik van bovenstaande polymorfe abstractie is een beetje lastig. Ten eerste moet elke instantiatie op de volgende manier expliciet gemaakt worden met behulp van het sleutelwoord new:

```
package S_GESORTEERDE_LIJSTEN is
        new GESORTEERDE_LIJSTEN(STRING, S_ORDENING);

package I_GESORTEERDE_LIJSTEN is
        new GESORTEERDE_LIJSTEN(INTEGER, I_ORDENING);
```

Elk van de geïnstantieerde packages heeft drie operaties die we kunnen aanroepen door de naam van het package vóór de naam van de operatie te plaatsen. Zo is S_GESORTEERDE_LIJSTEN.VOEG_TOE de naam van de operatie *voeg_ toe* van de package S_GESORTEERDE_LIJSTEN. We kunnen deze aanpak verbeteren door de operaties een andere naam te geven en overloading te gebruiken. De volgende twee declaraties veranderen de operaties VOEG_TOE van de twee geïnstantieerde packages weer in VOEG_TOE:

```
VOEG_TOE: procedure (W: INTEGER;
                     GL:I_GESORTEERDE_LIJSTEN.GESORTEERDE_LIJST)
            renames  I_GESORTEERDE_LIJSTEN.VOEG_TOE;

VOEG_TOE: procedure (W: STRING;
                     GL: S_GESORTEERDE_LIJSTEN.GESORTEERDE_LIJST)
            renames  S_GESORTEERDE_LIJSTEN.VOEG_TOE;
```

De programmeur kan nu de naam VOEG_TOE gebruiken als de naam van de polymorfe procedure.

Bij de meeste talen met polymorfe gegevensabstracties is deze lastige verzameling opdrachten voor het instantiëren en het opnieuw benoemen van operatoren niet nodig. Meestal kan de instantiatie automatisch gebeuren en worden de namen van de operatoren automatisch overloaded. Zo'n automatische gang van zaken is eenvoudiger te bereiken als de taal uitgaat van een semantisch standpunt ten opzichte van polymorfisme. Helaas maakt een dergelijke automatisering het systeem minder flexibel. Met de naamsveranderingen in de stijl van Ada kan men een bestaande gegevensabstractie gemakkelijk inpassen in een bestaand applicatieprogramma door packages en operaties eenvoudig andere namen te geven.

De stijl van ML

We bekijken nu Milners stijl van polymorfisme zoals die is neergelegd in de taal ML. In deze taal hoeven er geen declaraties te worden gegeven als die uit de context kunnen worden bepaald, hoewel we in ons voorbeeld alles zullen declareren. Doordat de algoritme voor typecontrole is gebaseerd op unificatie, kunnen polymorfe typen alleen type-parameters hebben; andere soorten parameters, zoals grenzen en operatoren, kunnen niet worden gebruikt. We moeten dus, net als bij PL/I, de procedure ORDENING meegeven aan de procedure LEGE_LIJST.

Om dezelfde lijn te volgen als in de vorige voorbeelden gebruiken we niet de syntaxis van ML en ook niet de geest van ML, die een waarde-gerichte implementatie zou aanmoedigen.

```
adt GESORTEERDE_LIJSTEN is

    type ORDENING (wtype: TYPE) is
        function (X, Y: wtype) return Boolean;

    type GESORTEERDE_LIJST (wtype: TYPE) is
        record
            VOLGORDE: ORDENING (wtype);
            WAARDEN: LIJST (wtype);
        end record;

    type LIJST (wtype: TYPE) is pointer
        record
            EERSTE: wtype;
            REST: LIJST (wtype);
        end record;

    LEGE_LIJST: procedure (GL: out GESORTEERDE_LIJST (wtype);
                           O: ORDENING (wtype)) is
    begin
        GL.VOLGORDE := O;
        GL.WAARDEN := null;
    end LEGE_LIJST;

    VOEG_TOE: procedure (W: wtype;
                         GL: in out GESORTEERDE_LIJST(wtype)) is
        ...
    end VOEG_TOE;

    DOORLOOP: procedure (GL: GESORTEERDE_LIJST (wtype);
                         PROC: procedure (X: wtype)) is
        ...
    end DOORLOOP;

end adt GESORTEERDE_LIJSTEN;
```

Het voorbeeld in de stijl van ML heeft veel gemeen met de stijl van PL/I. De gegevensstructuren lijken op elkaar, omdat in beide voorbeelden de ordeningsoperatie als deel van de waarde moet worden opgenomen in plaats van als deel van het gegevenstype, zoals in het voorbeeld in de stijl van Ada. In beide gebeurt de instantiatie impliciet. Maar er is een belangrijk semantisch verschil. Terwijl in de PL/I-aanpak geen typecontrole plaatsvindt, noch tijdens compilatie, noch tijdens het uitvoeren van het programma, vindt er in de ML-stijl volledige typecontrole plaats tijdens het compileren. We zien dat verschil het best als we letten op de manier waarop `IETS` en `type` in de beide voorbeelden wor-

den gebruikt. De ML-stijl is gebaseerd op een semantisch standpunt ten opzichte van polymorfisme en daarom wordt de typecontrole uitgevoerd voordat er instantiatie plaatsvindt. Dat is een verschil met de Ada-stijl, die berust op een syntactisch gezichtspunt ten opzichte van polymorfisme. Een combinatie van de laatste twee stijlen is ook mogelijk. Russell bijvoorbeeld heeft impliciete instantiatie, is gebaseerd op een semantisch uitgangspunt, en staat operatoren toe als parameter van polymorfe typen.

Opgaven

1. Breid de voorbeelden van gesorteerde lijsten uit met een operator *verwijder*.

2. Ontwerp en implementeer een type verzameling. Definieer de operaties *lege _verzameling*, *voeg_toe* en *lid*. Welke operatie(s) is (zijn) er nodig met het type als parameter? Hoe kan (kunnen) deze operatie(s) aan de gegevensabstractie worden doorgegeven in een ML-achtige of PL/I-achtige taal?

3. Welke typen kunnen we afleiden voor de variabelen in elk van de volgende expressies?

```
F(A, F(5, true))

G: function (X, Y) is
begin
   if Y = null then
         return X;
   else
         return APP(FIR(X), G(TA(X), Y));
   end if;
end G;
```

4. Beschouw het type van de volgende expressie:

```
X(X);
```

   Heeft x een type? Welke eigenschappen kunnen we afleiden voor het type van x?

Literatuur

Gehani en Gries (1977) bespreken problemen die met polymorfe typen optreden. In een aantal talen zijn verschillende vormen van polymorfisme gepro-

beerd, met name in Ada, Alphard (Wulf et al., 1976), EL1 (Wegbreit, 1974), ML (Gordon et al., 1978, 1979), HOPE (Burstall et al., 1980), Russell (Demers et al., 1978) en POLY (Harland, 1984). Mitchell en Wegbreit (1978) hebben Schemes ingevoerd als definities die met typen zijn geparametriseerd. Thatcher et al. (1979) en Ganzinger (1983) bespreken geparametriseerde typen in de context van algebraïsche specificaties.

Hindley (1969) gebruikte de door Robinson (1965) ontwikkelde unificatie-algoritme voor het bepalen van de hoofdtypering van expressies. Milner (1978) heeft dit idee verder ontwikkeld; het is gebruikt in ML en HOPE. Verder werk aan deductie van typen bij polymorfisme is gedaan door Damas en Milner (1982), Leivant (1983a), Mitchell (1984), McCracken (1984) en vele anderen.

De semantiek van polymorfisme heeft recentelijk belangstelling getrokken en wordt besproken door MacQueen en Sethi (1982), Coppo (1983), Taghva (1983), Reynolds (1983), Leivant (1983b) en nog recenter in de zesde POPL-bijeenkomst, 1984; zie Wand (1984), MacQueen et al. (1984) en Mitchell (1984). In zulk werk worden vaak lambdacalculus-modellen gebruikt, aangezien de lambdacalculus een eenvoudige, maar niet-triviale notatie levert voor het onderzoeken van verschillende semantieken voor polymorfisme; zie Barendregt (1981), Meyer (1982), Reynolds (1984) en Bruce en Meyer (1984). Burstall en Goguen (1977), Guttag en Horning (1983), Bert (1983) en Goguen (1984) bespreken de kwesties rond de formalisatie van eigenschappen van operaties die met type-parameters worden doorgegeven.

Deel IV
Specificatie van gegevenstypen

11

Specificaties

In dit hoofdstuk worden methoden voor het definiëren van gegevenstypen geïntroduceerd. Specificaties worden voor vele doeleinden geschreven. Op de eerste plaats zegt een specificatie precies hoe een gegevenstype zich gedraagt. Deze informatie stelt een programmeur in staat het gegevenstype te implementeren en stelt gebruikers in staat het te gebruiken. Op de tweede plaats verschaft een specificatie ons een middel om de correctheid van een implementatie van het gegevenstype na te gaan, door middel van testen of door formele verificatie. Op de derde plaats kan een specificatie een middel zijn voor het automatisch implementeren van gegevenstypen.

De twee belangrijkste eigenschappen van specificaties zijn precisie en communicatie. Specificaties moeten precies en ondubbelzinnig zijn, zodat het moeilijk is ze verkeerd te begrijpen of te interpreteren. Natuurlijke taal wordt gemakkelijk verkeerd begrepen en kan dubbelzinnig zijn, terwijl een formele taal precies en ondubbelzinnig kan worden gemaakt. Maar het is even belangrijk dat specificaties gemakkelijk te lezen zijn. Specificaties worden gebruikt in de communicatie tussen mensen; zowel de gebruiker als de implementator moeten de betekenis gemakkelijk kunnen begrijpen. Natuurlijke taal is gemakkelijk te lezen maar niet precies, de meeste formele talen zijn precies maar moeilijk te lezen. Deze twee tegenstrijdige doelen komen niet alleen bij specificaties van gegevenstypen voor; zij vormen een uitdaging voor de gehele informatica.

In nauw verband met specificaties van gegevenstypen staat het werk aan programmaspecificaties en formalismen voor het definiëren van de semantiek van

programmeertalen. Naast een aanpak op basis van natuurlijke taal zullen we ook kijken naar operationele, axiomatische, denotationele en algebraïsche methoden van specificatie. *Operationele semantiek* is een algemene term die verwijst naar één van de technieken voor het definiëren van programmeertalen.

Deze techniek berust op een procedurele beschrijving van de taal. Meestal wordt er een abstracte machine of interpretator gedefinieerd en wordt de programmeertaal gedefinieerd in termen van acties op deze abstracte machine. Dit is een natuurlijke aanpak, aangezien de definitie een getrouw beeld kan vormen van de feitelijke implementatie van de programmeertaal. Tot de voorbeelden van deze aanpak behoren de Vienna Definition Language (Lucas et al., 1968; Wegner, 1972) en SEMANOL (Andersen et al., 1976). Deze algemene methode is ook toepasbaar op programma-specificatie.

Floyd en Hoare hebben een logische (of axiomatische) aanpak geïntroduceerd die een implementatie-onafhankelijke manier vormt om het gedrag van een programma te specificeren. Deze methode definieert de semantiek van een programmeertaal met behulp van axioma's die het gedrag van constructies in de programmeertaal specificeren. Een programma wordt gespecificeerd door axioma's op de invoer en de uitvoer. De consistentie van deze axioma's met het programma kan worden bewezen met behulp van de axiomatische definitie van de programmeertaal. De door Scott ontwikkelde denotationele methode geeft een wiskundige beschrijving van programmeertalen en is gebaseerd op recursieve domeinvergelijkingen. Bij deze methode worden er functies geconstrueerd die programma's afbeelden in een semantisch domein. Een dergelijke methode kan ook worden toegepast voor het definiëren van programma's en gegevensabstracties. De algebraïsche methode lijkt door het gebruik van axioma's op de axiomatische methode.

In dit hoofdstuk geven we voorbeelden van al deze methoden. Later ontwikkelen we de algebraïsche aanpak dan helemaal en geven we grotere voorbeelden van het gebruik van die methode voor het definiëren van gegevenstypen.

11.1 Voorbeeld: specificatie van een editor

Als voorbeeld kiezen we een tekst-editor met een paar eenvoudige commando's. De editor bewerkt een file, dat wil zeggen een lineaire serie records. Records worden als gegeven gegevensobjecten beschouwd en niet verder gedefinieerd. In feite kan het editor-voorbeeld worden gezien als een polymorf type waarvan *record* een type-parameter is. Editor-commando's veranderen de file door het toevoegen, vervangen en verwijderen van records. De commando's werken op het *actuele record*, een bepaald record in de file. In de Nederlandstalige en de operationele specificatie is de file een impliciete parameter van elk editor-commando. In de functionele en de algebraïsche specificatie is de file een expliciete parameter.

11.2 Specificaties in het Nederlands

De Nederlandstalige specificatie van de editor ziet eruit als een lijst commando's met elk een korte beschrijving bestaande uit één zin. Om de beschrijving van elke operatie eenvoudig te houden geven we de uitzonderingen apart.

Operaties

NieuweFile: Initialiseer; creëer een lege file zonder actueel record.

VoegToe(NieuwRecord): Voeg NieuwRecord toe na het actuele record en maak NieuwRecord tot het actuele record.

Vervang(NieuwRecord): Vervang het actuele record door NieuwRecord.

Verwijder: Verwijder het actuele record en maak het volgende record tot het actuele record.

Vooruit: Maak het volgende record tot het actuele record.

Terug: Maak het vorige record tot het actuele record.

Kopieer(m,n): Kopieer m records (te beginnen met het actuele record) achter record n.

Uitzonderingen

Operatie	*Conditie*	*Actie*
Verwijder	geen actueel record	geen actie
Terug	geen actueel record	geen actie
Vervang	geen actueel record	geen actie
Vooruit	geen actueel record	het eerste record wordt het actuele record
Verwijder	het actuele record is het laatste record	het vorige record wordt het actuele record
Vooruit	het actuele record is het laatste record	geen actie
Terug	het actuele record is het eerste record	geen actueel record

11.3 Operationele specificaties

Operationele specificaties implementeren de gegevensabstractie in een hogere taal en letten niet op efficiëntie. Deze implementatie moet niet worden gezien als een model van de echte implementatie, maar alleen als een model van de gegevensabstractie. In de volgende specificatie wordt een file gerepresenteerd als een array (van onbepaalde lengte) van records.

```
File: array [1.. ] of record;
FileLengte: integer;
Actueel: integer;

NieuweFile:
        FileLengte = 0;
        Actueel = 0;

VoegToe(NieuwRecord):
        for j from FileLengte to Actueel+1 by –1 loop
```

```
            File[j+1] = File[j];
            end loop;
        File[Actueel+1] = NieuwRecord;
        FileLengte = FileLengte + 1;
        Actueel = Actueel + 1;

Vervang(NieuwRecord):
        if Actueel ≠ 0 then
            File[Actueel] = NieuwRecord;

Verwijder:
        if Actueel ≠ 0 then
            for j from Actueel to FileLengte–1 loop
                File[j] = File[j+1];
                end loop;
            FileLengte = FileLengte – 1;
            if Actueel > FileLengte then
                Actueel = FileLengte;

Vooruit:
        if Actueel ≠ FileLengte then
            Actueel = Actueel + 1;

Terug:
        if Actueel ≠ 0 then
            Actueel = Actueel – 1;

Kopieer(m,n):
        ...
```

11.4 Logische specificaties

Logische specificaties gebruiken uitspraken of *asserties* over invoer en uitvoer, geschreven in predikatencalculus, om de condities te beschrijven die gelden vóór en na de uitvoering van opdrachten, procedures en programma's. Als de invoer-assertie waar is voordat het programma wordt uitgevoerd, moet na het uitvoeren van het programma de uitvoer-assertie waar zijn. Meestal worden er variabelen uit het programma gebruikt om de invoer- en uitvoerwaarden te representeren. In de volgende specificatie is de variabele File een flexibele array

van records (lopend vanaf 1) en is Actueel een index bij File die naar het actuele record wijst (evenals in de vorige specificatie). Op dezelfde manier representeert FileLengte het actuele aantal records in de file. Evenals in de vorige specificatie moeten de variabelen File, Actueel en FileLengte worden beschouwd als een abstract model van de te editeren file. $File_0$ en $Actueel_0$ zijn de waarden van de variabelen File en Actueel voordat een editor-commando wordt uitgevoerd. De invoer-assertie is voor alle editorcommando's gelijk en wordt dan ook als invariante assertie aangegeven; daarom wordt voor elk commando alleen de uitvoer-assertie gegeven.

File: array [1..] of record;
FileLengte: integer;
Actueel: integer;
NieuwRecord: record — parameter van VoegToe en Vervang

Invariante assertie: $0 \leq Actueel \leq FileLengte$

Uitvoer-asserties

NieuweFile:

$$Actueel = 0 \land FileLengte = 0$$

VoegToe:

$$\begin{aligned}
&Actueel = Actueel_0 \land FileLengte = FileLengte_0 + 1\\
&\quad \land (\forall j)((j < Actueel \rightarrow File(j) = File_0(j))\\
&\qquad \land (j = Actueel \rightarrow File(j) = NieuwRecord)\\
&\qquad \land (j > Actueel \rightarrow File(j) = File_0(j-1)))
\end{aligned}$$

Vervang:

$$\begin{aligned}
&Actueel = Actueel_0 \land FileLengte = FileLengte_0\\
&\quad \land (\forall j)((j = Actueel \rightarrow File(j) = NieuwRecord)\\
&\qquad \land (j \neq Huidig \rightarrow File(j) = File0(j)))
\end{aligned}$$

Verwijder:

$$\begin{aligned}
&(Actueel = Actueel_0 = 0 \land File = File_0\\
&\land FileLengte = FileLengte_0)\\
&\lor (((Actueel_0 = FileLengte \land Actueel = Actueel_0 - 1)\\
&\qquad \lor (Actueel_0 \neq FileLengte_0 \land Actueel = Actueel_0))\\
&\quad \land FileLengte = FileLengte_0 - 1\\
&\quad \land (\forall j)((j < Actueel_0 \rightarrow File(j) = File_0(j))\\
&\qquad \land (j \geq Actueel_0 \rightarrow File(j) = File_0(j+1))))
\end{aligned}$$

Vooruit:

$$\text{File} = \text{File}_0 \wedge \text{FileLengte} = \text{FileLengte}_0$$
$$\wedge\ (\text{Actueel} = \text{Actueel}_0 = \text{FileLengte}_0$$
$$\vee\ (\text{Actueel}_0 \neq \text{FileLengte}_0 \wedge \text{Actueel} = \text{Actueel}_0 + 1))$$

Terug:

$$\text{File} = \text{File}_0 \wedge \text{FileLengte} = \text{FileLengte}_0 \wedge (\text{Actueel} = \text{Actueel}_0 = 0$$
$$\vee\ (\text{Actueel}_0 \neq 0 \wedge \text{Actueel} = \text{Actueel}_0 - 1))$$

Kopieer(m,n):

$$\text{File} = \text{File}_0 \wedge \text{FileLengte} = \text{FileLengte}_0 \wedge \text{Actueel} = \text{Actueel}_0$$
$$\wedge\ (n > \text{FileLengte}_0 \vee \text{Actueel}_0 + m > \text{FileLengte}_0 \vee \text{Actueel}_0 = 0)$$
$$\vee\ (n \leq \text{FileLengte}_0 \wedge \text{Actueel}_0 + m \leq \text{FileLengte}_0 \wedge \text{Actueel}_0 \neq 0$$
$$\wedge\ \text{FileLengte} = \text{FileLengte}_0 + m$$
$$\wedge\ (\text{Actueel} = \text{Actueel}_0 \wedge n \geq \text{Actueel}$$
$$\vee\ \text{Actueel} = \text{Actueel}_0 + m \wedge n < \text{Actueel})$$
$$\wedge\ (\forall j)((j \leq n \rightarrow \text{File}(j) = \text{File}_0(j))$$
$$\wedge\ (n < j \leq n + m \rightarrow \text{File}(j) = \text{File}_0(\text{Actueel} + j - n + 1))$$
$$\wedge\ (j > n + m \rightarrow \text{File}(j) = \text{File}_0(j + m))))$$

11.5 Functionele specificaties

Functionele specificaties beschrijven het gedrag van functies door middel van een wiskundige functie die de relatie tussen de in- en uitvoer precies beschrijft. In tegenstelling tot operationele specificaties, die een bepaalde methode beschrijven om de uitvoer uit de invoer te *berekenen*, behoeven functionele specificaties slechts de uitvoer in termen van de invoer te beschrijven. In functionele specificaties worden voor het representeren van objecten traditionele wiskundige constructies gebruikt, zoals verzamelingen, functies en sequences. Een programma wordt beschouwd als een functie van de invoer naar de uitvoer en wordt ook zo beschreven. Voor het aangeven van een functie wordt de lambda-calculus gebruikt, en de expressie

$$a \rightarrow b, c$$

betekent 'if a then b else c'. De verzameling van alle (continue) functies van verzameling A naar verzameling B wordt voorgesteld als '$[A \rightarrow B]$', het Carthesisch produkt als '$A \times B$' en tupels en sequences worden voorgesteld als lijsten van waarden met hoekige haakjes eromheen en gescheiden door puntkomma's

(bijvoorbeeld $<a; b> \in A \times B$). In de specificatie van de editor wordt elk commando beschouwd als een functie met de file als expliciete parameter. De verzameling FILES is een twee-tupel waarvan het eerste element het actuele record aangeeft en het tweede element een functie van de natuurlijke getallen naar RECORDS, hetgeen een record kan zijn of *void*.

N = verzameling van alle natuurlijke getallen
RECORDS = verzameling van alle records plus *void*
FILES = N × [N → RECORDS]

m, n, j, p ∈ N
f ∈ [N → RECORDS]
r ∈ RECORDS

NieuweFile: → Files
NieuweFile = $< 0, \lambda j.\mathit{void} >$

Vervang: RECORDS × FILES → FILES
$\text{Vervang}(r, < p;f >) = < p; \lambda j.(p = 0) \to f(j),(p = j) \to r, f(j) >$

VoegToe: RECORDS × FILES → FILES
$\text{VoegToe}(r, <p;f>) = <p + 1; \lambda j.(j \le p) \to f(j),(j = p + 1) \to r, f(j-1) >$

Verwijder: FILES → FILES
$\text{Verwijder}(< p;f >) = (p = 0) \to < p; f >, < (f(p + 1) = \mathit{void}) \to p - 1, p;$
$\lambda j.(j < p) \to f(j), f(j + 1) >$

Vooruit: FILES → FILES
$\text{Vooruit}(< p;f >) = < (f(p + 1) = \mathit{void}) \to p, p + 1; f >$

Terug: FILES → FILES
$\text{Terug}(< p;f >) = < (p = 0) \to p, p - 1; f >$

Kopieer: N × N × FILES → FILES
$\text{Kopieer}(m, n, < p;f >) = (f(n) = \mathit{void} \text{ of } (f(p + m - 1) = \mathit{void}) \to < p;f >,$
$< (n < p) \to p + n, p;$
$\lambda j.(j < n) \to f(j), (n \le j \le n + m) \to f(p + j - n + 1), f(j + m) >$

11.6 Algebraïsche specificaties

Onze volgende specificatie volgt de algebraïsche methode, die in hoofdstuk 13 veel gedetailleerder wordt beschreven. In de algebraïsche methode worden gegevenstypen als algebra's beschouwd en schrijft men voor het specificeren van een type de axioma's op die de algebra beschrijven. De axioma's zijn gemakkelijker te beschrijven als er een canonieke vorm kan worden gevonden. De canonieke vorm is een verzameling expressies die alle mogelijke waarden van het gegevenstype specificeren, en wel zo dat twee ongelijke expressies ook altijd voor ongelijke waarden staan. Een canonieke vorm voor het voorbeeld van de teksteditor is

Terug(...Terug(VoegToe(r_m, ...VoegToe(r_1, NieuweFile)...)

waarin r_1, r_2, ..., r_m de rij records is waaruit de file bestaat en waarin het aantal keer Terug het actuele record aangeeft (gerekend vanaf het laatste record). Er is nog een extra *verborgen* operatie M nodig voor het specificeren van dit gegevenstype. M kan worden gezien als een manier om een file als twee delen te beschouwen. Het eerste deel bestaat uit alle records die voorafgaan aan het actuele record, plus het actuele record zelf. Het tweede deel bestaat uit alle records die op het actuele record volgen, maar in omgekeerde volgorde. Dan is een andere canonieke vorm van de file:

M(VoegToe(r_n, VoegToe(r_n-1, ..., VoegToe(r_1, NieuweFile)...),
VoegToe(r_{n+1}, VoegToe(r_{n+2}, ..., VoegToe(r_m, NieuweFile)...)

waarin de file bestaat uit de rij r_1, r_2, ..., r_m en waarin r_n het actuele record is, met $0 \leq n \leq m$. Merk op dat de axioma's voor de operator Kopieer achterwege zijn gelaten als oefening voor de lezers die menen dat het gebruik van algebraïsche axioma's eenvoudig is.

Gegevenstypen

FILES = gegevenstype File
RECORDS = gegevenstype Record
N = gegevenstype Getal

Operatoren

NieuweFile: $\rightarrow$ FILES
Voegtoe: RECORDS $\times$ FILES $\rightarrow$ FILES
Vervang: RECORDS $\times$ FILES $\rightarrow$ FILES
Verwijder: FILES $\rightarrow$ FILES
Vooruit: FILES $\rightarrow$ FILES
Terug: FILES $\rightarrow$ FILES
Kopieer: N $\times$ N $\times$ FILES $\rightarrow$ FILES
M: FILES $\times$ FILES $\rightarrow$ FILES

Axioma's

NieuweFile = M(NieuweFile, NieuweFile)

VoegToe(r, M(x, y)) = M(VoegToe(r, x), y)

Vervang(r, M(VoegToe(s, x), y)) = M(VoegToe(r, x), y)
Vervang(r, M(NieuweFile, y)) = M(NieuweFile, y)

Verwijder(M(VoegToe(r, x), y)) = M(x, y)
Verwijder(M(NieuweFile, y)) = M(NieuweFile, y)

Vooruit(M(x, VoegToe(r, y))) = M(VoegToe(r, x), y)
Vooruit(M(x, NieuweFile)) = M(x, NieuweFile)

Terug(M(VoegToe(r, x), y)) = M(x, VoegToe(r, y))
Terug(M(NieuweFile, y)) = M(NieuweFile, y)

11.7 Vergelijking van specificatiemethoden

Het vergelijken van specificaties is niet gemakkelijk omdat het gedeeltelijk op subjectieve criteria berust. Marcotty et al. (1976) en Donahue (1976) hebben vergelijkingen tussen semantische formalismen gemaakt. Behalve de specificatie in het Nederlands legt iedere specificatiemethode een bepaald uitgangspunt op. Nederlandstalige specificaties zijn gemakkelijk te schrijven omdat ze aan allerlei uitgangspunten en graden van detaillering zijn aan te passen.

Deze flexibiliteit maakt de natuurlijke taal tot een wendbare manier om bedoelingen over te brengen; het Nederlands is alleen onvoldoende als men precies wil zijn.

De andere specificaties leggen de gebruikers een bepaald uitgangspunt op. Operationele specificaties beschrijven alles in termen van gegevensstructuren en algoritmen. Bij die aanpak zijn meestal meer details nodig dan men zou willen. Asserties laten een breed scala van uitgangspunten toe voor het beschrijven van bepaalde objecten of gegevensstructuren. Meestal worden er variabelen en gegevenstypen uit het programma gebruikt, omdat de verificatie van het programma dan gemakkelijker is. Asserties dwingen de gebruiker het programma vanuit een ongebruikelijke gezichtshoek te bekijken. Het programma wordt gezien als een object waarover asserties worden gemaakt. De specificaties leggen feiten vast betreffende het gedrag van het programma.

Ook functionele specificaties laten allerlei beschrijvingen toe. Zo kan een array bijvoorbeeld worden beschouwd als een verzameling geordende paren, als een sequence of als een functie. De functionele specificatiemethode legt als uitgangspunt op dat een programma een wiskundige functie is. De algebraïsche methode legt het uitgangspunt op van een formeel systeem met axioma's en expressies. Axioma's worden beschouwd als regels voor het manipuleren van bomen van operatoren. Vanuit deze gezichtshoek worden waarden van nieuwe gegevenstypen uitsluitend als klassen van expressies gezien (of als parseerbomen van operatoren).

Al met al worden bij de vier formele methoden programma's op zeer verschillende manieren gezien. Operationele specificaties behandelen programma's als algoritmen. Logische specificaties behandelen programma's door asserties op te stellen over de in- en uitvoer. Functionele specificaties behandelen programma's als wiskundige functies. En algebraïsche specificaties behandelen programma's als door axioma's beschreven algebra's.

Een specificatie is *volledig* als de uitvoer van een programma of functie voor elke legitieme invoer uit de specificatie kan worden bepaald. Een specificatie is *consistent* (ondubbelzinnig) als voor elke invoer voor het programma een unieke uitvoer wordt gespecificeerd. Hoewel volledigheid en consistentie eigenschappen van een bepaalde specificatie zijn, heeft de specificatietaal een grote invloed op het gemak waarmee volledige en consistente specificaties te bereiken zijn en op het gemak waarmee is vast te stellen of een bepaalde specificatie volledig of consistent is.

Als Nederlandstalige specificaties niet zorgvuldig zijn geschreven is het vrijwel onmogelijk vast te stellen of ze volledig zijn. Het dubbelzinnige karakter van de taal en onze neiging meer in een zin te lezen dan er staat, maken het onwaarschijnlijk dat in natuurlijke taal ooit specificaties te schrijven zullen zijn die gemakkelijk als volledig herkenbaar zijn. In de editor-specificaties is er bijvoorbeeld een lijst van uitzonderingen, die eigenlijk geen uitzonderingen zijn, maar eerder toevoegingen om de specificatie van de operatoren volledig te maken. Als één van die uitzonderingen zou ontbreken, zou de specificatie onvolledig zijn. Hoe groot is de kans dat de ontbrekende uitzondering gemakkelijk op te sporen zou zijn? Het ontbreken zou worden ontdekt als er een grondige analyse van de editor-specificaties zou worden gemaakt. Ter illustratie zijn er enkele uitzonderingen met opzet weggelaten. Die uitzonderingen zijn gemakkelijk op te sporen door middel van een analyse van de functionele specificatie van Kopieer, maar niet als men gewoon de Nederlandse specificatie aan een onderzoek onderwerpt. Meestal is het Nederlands (of een andere natuurlijke taal) ongeschikt voor het geven van volledige specificaties. Om dezelfde redenen is consistentie al even moeilijk vast te stellen. Als er bijvoorbeeld extra uitzonderingen zouden worden toegevoegd, die in tegenspraak zouden zijn met de algemene regels voor de operatoren of met de andere uitzonderingen, wat zou dan voorrang hebben?

Een operationele definitie geeft ons een algoritmische manier om de resultaten van een programma of functie te bepalen. Net als echte programma's zijn operationele specificaties meestal volledig en consistent doordat er maar één manier is om een programma te interpreteren. Men kan soms een operationele specificatie tegenkomen waar gaten in zitten ten gevolge van *fouten bij het uitvoeren*, zoals delen door nul, of logische fouten die tot eindeloze herhaling leiden en daardoor geen antwoord leveren.

Stel dat de array van records in de operationele specificatie van de editor een bovengrens zou hebben. Wat zou er dan gebeuren als er een index was die boven die bovengrens uit zou komen, zodat er een fout 'index buiten de grenzen' zou optreden? We zouden dan tot de conclusie moeten komen dat de operationele specificatie onvolledig zou zijn.

Logische specificaties zijn er berucht om dat ze onvolledig kunnen zijn (zie Gerhart en Yelowitz, 1976). Het klassieke voorbeeld is de specificatie van het sorteren van een array waarin alleen wordt gesteld dat de resulterende array in stijgende volgorde staat. Die specificatie is onvolledig omdat de relatie tussen de uitvoerarray en de oorspronkelijke, niet gesorteerde invoerarray niet wordt

gelegd. Sommige van de oorzaken waardoor asserties problemen geven, zijn gemakkelijk op te sporen. Beschouw de assertie '($P(x)$ en $A(x)$) of $B(x)$', waarin de predikaten A en B de uitvoerwaarden specificeren en predikaat P een of andere voorwaarde specificeert. Het is de bedoeling dat A aan bod komt als P waar is en B als P onwaar is. Maar als het zo wordt opgeschreven, kan B altijd aan bod komen. De correcte assertie luidt '($P(x)$ en $A(x)$) of (niet $P(x)$ en $B(x)$)'. Men vergeet gemakkelijk 'niet $P(x)$' toe te voegen. Inconsistentie komt bij asserties voor, maar als het wel voorkomt is het gevolg dat de uitvoerassertie altijd onwaar is, en daarvoor kan geen programma worden geschreven.

In tegenstelling tot de logische methode specificeert de functionele methode de uitvoerwaarden direct in termen van de invoerwaarden, in plaats van indirect met asserties. Net als operationele specificaties leiden functionele specificaties niet gauw tot inconsistente specificaties. Bovendien zijn functionele specificaties niet gauw onvolledig, omdat men de uitvoer specificeert voor een willekeurige waarde van de invoer. Werden bij het voorbeeld van de te sorteren array de invoerwaarden in de asserties vergeten, dan zal dat in operationele of functionele specificaties niet gebeuren.

Het is moeilijk de volledigheid van een algebraïsche specificatie in het algemeen vast te stellen (Guttag, 1975). Een goed voorbeeld daarvan is te proberen de algebraïsche axioma's voor het gegevenstype verzameling te beschrijven. Men kan steeds met nieuwe axioma's aankomen die verzamelingen beschrijven, en na een tijdje is men er niet zeker van of er nog meer axioma's nodig zijn. De gebruikelijke methode om bij het completeren van een verzameling axioma's te helpen is een canonieke vorm voor alle objecten te bedenken (met behulp van constructor-operatoren) en dan axioma's te definiëren die elke expressie met andere operatoren terugbrengen tot een canonieke vorm. Inconsistentie treedt op als de axioma's leiden tot het resultaat dat twee objecten equivalent zijn die verschillend hadden moeten zijn. Dit probleem wordt soms ontdekt als bij het gegevenstype Boolean *false* equivalent wordt met *true*.

Opgaven

1. Beschouw het voorbeeld van de editor. Breid het uit met de operatie *eerste*, die naar het eerste record van de file gaat, en met *verwissel*, die het actuele record verwisselt met het volgende. Voeg deze twee operaties aan elk van de vijf specificaties toe. Vergelijk de vijf gemaakte uitbreidingen. Welke specificaties waren het gemakkelijkst te schrijven?

2. De Nederlandse specificatie is onvolledig. Geef een opsomming van dingen die in de Nederlandse specificatie ontbreken. Het volgende is een aanwijzing om te ontdekken welke informatie er bij de operator Kopieer ontbreekt. Wat gebeurt er als de file minder dan n records bevat? Wat gebeurt er als er minder dan m records komen na het actuele record? Verandert het actuele record na een commando Kopieer? Loopt de index van de file vanaf 1 of vanaf 0?

12

De wiskunde van gegevenstypen

In hoofdstuk 1 hebben we bij het definiëren van gegevenstypen een mengsel van Nederlands en wiskunde gebruikt. Maar we hebben een betere en preciezere definitie nodig om problemen rond typecontrole, polymorfisme, specificatie en correctheid op een consistente manier te kunnen bespreken. Evenals in andere vakken verschaft een wiskundige fundering ons hier de juiste hulpmiddelen die ons in staat stellen gegevenstypen te beschrijven en streng te definiëren. Zo'n fundering kan inzicht verschaffen in de donkere hoeken van gegevenstypen en kan opheldering geven over de syntactische en semantische aspecten van polymorfisme, typecontrole en correctheid. Er zijn voor gegevenstypen vele mathematische modellen voorgesteld. In dit boek bestuderen we de meest gebruikelijke methode, soms de algebraïsche methode genoemd, in detail. Deze methode kan worden gekarakteriseerd als *operator-gericht*, omdat de nadruk ligt op de operatoren van een type. Een andere aanpak is gebaseerd op *domeinen*, hetgeen gewoon verzamelingen betekent. Deze tweede methode kan worden gekarakteriseerd als *waarde-gericht*, omdat daarbij vooral aandacht wordt gegeven aan de verzameling waarden van een gegevenstype. Operaties op de waarden komen pas op de tweede plaats. Ondanks het verschil in uitgangspunt komen beide methoden ongeveer tot hetzelfde resultaat, namelijk dat een gegevenstype een *algebra* is – een verzameling waarden en een verzameling operaties op die waarden.

12.1 Algebra's en gegevenstypen

Voordat we de wiskunde van de algebraïsche methode ontwikkelen geven we eerst een intuïtief overzicht van deze aanpak. De algebraïsche methode is gericht op de operaties van een gegevenstype; we zullen daarom eerst de operaties bestuderen en dan de waarden. Een operatie wordt gedenoteerd (aangeduid) door een operatorsymbool; tot de operaties behoren ook nul-aire operatoren oftewel constanten. Als een collectie operatorsymbolen gegeven is, kunnen we de verzameling van alle mogelijke expressies construeren, die de *woord-algebra* wordt genoemd. De woordalgebra definieert het gebied van alle mogelijke waarden van het gegevenstype. Om de semantiek te beschrijven behoeven we alleen te beschrijven welke expressies dezelfde waarde hebben. Dat doen we door een congruentie-relatie op de woordalgebra te specificeren.

12.2 Signaturen: de syntaxis van gegevenstypen

Een *soort* is de naam van een gegevenstype. In de algebraïsche methode wordt onderscheid gemaakt tussen de soort (een naam of symbool) en een gegevenstype (een algebra). *Operatoren*, misschien juister *operatiesymbolen* genoemd, zijn symbolen die worden gebruikt om operaties te representeren. De *ariteit* van een operator specificeert de soort van alle operanden en de soort van de waarde die door de operator wordt afgeleverd. *Constanten* zijn operatoren zonder operanden, dat wil zeggen constanten zijn nul-aire operatoren. De notatie die wordt gebruikt om de ariteit aan te geven van een operator σ die *n* operanden heeft van de soorten $s_1, s_2, ..., s_n$ en die een waarde van soort *s* aflevert, is:

$$\sigma: s_1 \times s_2 \times ... \times s_n \rightarrow s$$

De notatie suggereert bewust dat de door σ gerepresenteerde operatie een functie is, maar denk erom dat de operator σ een symbool is en geen functie. De ariteit wordt als een syntactisch begrip beschouwd dat bij een symbool hoort, niet als een semantisch begrip dat bij een functie hoort.

Zij *S* een verzameling soorten. Een *S-soortige signatuur* (of als *S* vanzelfsprekend of onbelangrijk is kortweg *signatuur*) is een verzameling operatoren met ariteiten waarin alleen soorten uit *S* voorkomen.

Een voorbeeld van een signatuur met $S = \{\text{int}\}$ met constanten 0 en 1 en de binaire operatoren + en * is:

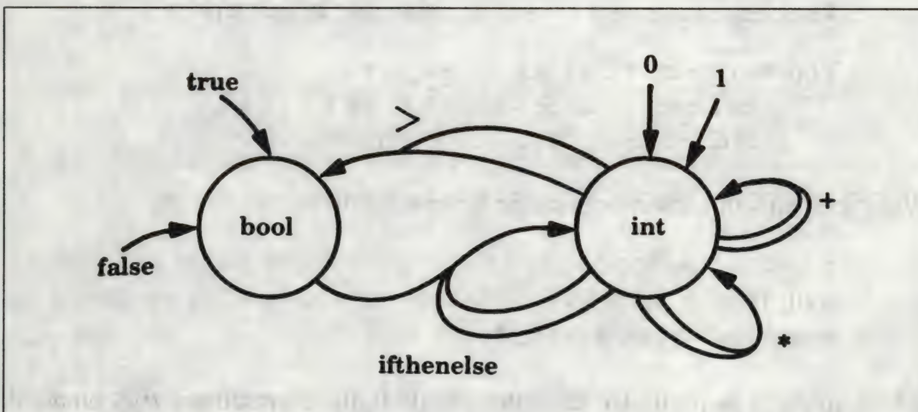
0	: → int
1	: → int
+	: int × int → int
*	: int × int → int

Dit wordt een 1-soortige signatuur genoemd, omdat er maar één soort in voorkomt. Veelsoortige signaturen zijn in de informatica gebruikelijker. Beschouw bijvoorbeeld de volgende, meer realistische signatuur die de soorten int en bool gebruikt:

true	: → bool
false	: → bool
0	: → int
1	: → int
+	: int × int → int
*	: int × int → int
>	: int × int → bool
ifthenelse	: bool × int × int → int

De laatste operator heeft drie operanden, niet alle van dezelfde soort.

Een andere, meer visuele beschrijving van signaturen wordt geleverd door het signatuurdiagram. Cirkels staan voor soorten en pijlen met meerdere staarten staan voor de ariteit van de operanden. Het signatuurdiagram van bovenstaande signatuur is te zien in figuur 12-1.



Figuur 12-1 Signatuurdiagram.

Bij het gebruik van zulke vertrouwde voorbeelden is het verleidelijk te veel betekenis in de signatuur te leggen. Het is verleidelijk de soort 'int' te zien als de integers, de operator '1' als één en de operator '+' als de optelling. Maar een signatuur beschrijft uitsluitend operatoren en ariteiten, geen operaties of waarden van een type. Een signatuur beschrijft de syntaxis van een gegevenstype, niet de semantiek.

Bij het formuleren van een wiskundige fundering voor gegevenstypen is het soms handig bepaalde uitzonderlijke gevallen buiten beschouwing te laten. Sommige operaties (zoals *maximum* en *minimum*) kunnen bijvoorbeeld een wisselend aantal argumenten hebben; andere operatoren kunnen meer dan één waarde afleveren, misschien via zij-effecten, zoals de operatie *pop*, die een stack verandert en de bovenste waarde aflevert. Een signatuur houdt met deze voorbeelden niet op directe wijze rekening. Een indirecte beschrijving is wel mogelijk, in het eerste geval door max(a,b,c) te beschouwen als een afkorting van max(a, max(b,c)) of max te beschouwen als een functie met één argument dat gevormd wordt door een lijst integers. In het tweede geval kan de operatie *pop* worden gesplitst in twee operaties, één voor elk van de beide verschillende waarden die worden afgeleverd.

12.3 Termen

Expressies die met operatoren zijn opgebouwd heten *termen*. Een constante is op zichzelf een term. Alle andere operatoren hebben operanden die ook weer termen zijn. De formele notatie voor termen wordt als volgt gedefinieerd:

Voor een constante $\sigma : \rightarrow s$ is de expressie 'σ' een term.

Voor een operator $\sigma : s_1 \times s_2 \times \ldots \times s_n \rightarrow s$
en termen t_i van de soorten s_i (voor $1 \leq i \leq n$)
is de expressie 'σ (t_1, ..., t_n)' een term.

Voorbeelden van termen volgens de bovenstaande signatuur zijn:

1
+ (0, 1)
ifthenelse(> (0, 1), + (1, 1), 1)

Merk op dat '+(> (0, 0), 0)' en 'ifthenelse(0, 0, 0)' *geen* termen zijn, omdat de definitie van term ook typecontrole omvat.

Uit een signatuur kan gemakkelijk een grammatica worden opgebouwd die alle termen genereert. Elke soort wordt vervangen door een niet-terminaal symbool. Elke operator

$$\sigma : s_1 \times ... \times s_n \rightarrow \text{s (met } n > 0)$$

wordt vervangen door de produktieregel

$$s \rightarrow \sigma(s_1, ..., s_n)$$

en elke constante '$\sigma : \rightarrow s$' wordt vervangen door de produktieregel

$$s \rightarrow \sigma$$

De grammatica die op deze manier uit de voorafgaande signatuur wordt gevormd is:

bool	→	true
bool	→	false
int	→	0
int	→	1
int	→	+ (int, int)
int	→	* (int, int)
bool	→	> (int, int)
int	→	ifthenelse(bool, int, int)

Merk op dat er zonder constanten geen termen kunnen bestaan.

Omdat de formele notatie voor termen niet altijd leesbaar is, zullen we vaak zonder aankondiging teruggrijpen naar een informele notatie. De operator '+' is bijvoorbeeld meestal een infix-operator, waardoor het leesbaarder is als in plaats van '+ (0, 1)' geschreven wordt '(0 + 1)'. Op dezelfde manier is 'if true then 0 else 1' gebruikelijker dan 'ifthenelse(true, 0, 1)'. We zullen ons nog andere syntactische vrijheden veroorloven, zoals '0 + 1*0' in plaats van '(0 + (1*0))'. Merk op dat '0 + 0 + 0' dubbelzinnig is omdat 0 en + niet meer zijn dan syntactische symbolen en de expressie op twee manieren kan worden geparseerd, die mogelijk tot twee verschillende betekenissen kunnen leiden. (Associativiteit is een eigenschap van operaties, niet van operatoren.) In zulke gevallen zullen we voor het oplossen van dubbelzinnigheden terugvallen op de gebruikelijke prioriteitsregels. We zijn vooral geïnteresseerd in de syntactische structuur van een term, niet in de syntactische buitenkant. In alle voorkomende

gevallen gaan we uit van een ondubbelzinnige parsering van informeel genoteerde termen. Termen moeten worden gezien als parseerbomen in plaats van rijtjes symbolen.

Zij Σ een S-soortige signatuur. De verzameling van alle termen van soort s zal worden aangeduid met *termen* (s,Σ). Dit is de taal die wordt voortgebracht door de grammatica met s als startsymbool. Deze verzameling termen, de verzameling van alle expressies van soort s, wordt soms het Herbrand-universum genoemd. Dat speelt een belangrijke rol bij algebraïsche specificaties. Als de signatuur vanzelfsprekend is, gebruiken we de afkorting *termen*(s) om de verzameling van alle termen van soort s aan te duiden.

12.4 Polymorfe typen

We zijn vaak geïnteresseerd in een hele klasse van typen met gelijksoortige operaties. We kunnen bijvoorbeeld de drie typen *verzameling van integers*, *verzameling van tekens* en *verzameling van verzamelingen van integers* beschouwen als instanties van het polymorfe type *verzameling*(Z), uit te spreken als 'verzameling van Z'. Om de wiskunde in dit hoofdstuk te vereenvoudigen kiezen we het eenvoudige syntactische uitgangspunt ten opzichte van polymorfisme. De specificatie van polymorfe typen wordt daardoor gelijk aan die van niet-polymorfe typen. Het volgende is een voorbeeld van een signatuur van het polymorfe type verzameling:

lege_verzameling	: → verzameling(Z)
voeg-toe	: Z × verzameling(Z) → verzameling(Z)
vereniging	: verzameling(Z) × verzameling(Z) → verzameling(Z)
lid	: Z × verzameling(Z) → bool

Elke operatorsymbool is dubbelzinnig (of overloaded), aangezien het verschillende operatoren kan representeren. Als de dubbelzinnigheid niet op grond van de context kan worden opgelost, zullen we aan de operator een index toevoegen; zo zal $\textit{lege_verzameling}_{int}$ een constante zijn van de soort *verzameling*(int).

12.5 Algebra's: de semantiek van gegevenstypen

Zij Σ een S-soortige signatuur. Dan is een Σ-*algebra* α een familie van verzamelingen

$$A = \{A_s \mid s \in S\}$$

en een verzameling operaties

$$\{\sigma_\alpha \mid \sigma \in \Sigma\}$$

zodanig dat als $\sigma : s_1 \times ... \times s_n \rightarrow$ s, dan geldt

$$\sigma_A : A_{s_1} \times ... \times A_{s_n} \rightarrow A_s$$

A_s wordt de *drager* voor de soort s genoemd. Merk op dat σ een operator is en σ_α een operatie. De operator σ wordt bijvoorbeeld gebruikt om termen van soort s te construeren uit termen van de soorten s_1 tot en met s_n, terwijl σ_A een functie is van de verzamelingen A_{s_1} tot en met A_{s_n} naar de verzameling A_s. Hoewel we later in dit hoofdstuk ons concept van een gegevenstype zullen verfijnen, kunnen we elke A_s nu beschouwen als de verzameling waarden van type s.

Om het onderstaande voorbeeld te vereenvoudigen gebruiken we een signatuur met maar één soort. Zij Σ de eenvoudige signatuur die we eerder hebben gebruikt:

0	: → int
1	: → int
+	: int × int → int
*	: int × int → int

Beschouw nu de volgende zes Σ-algebra's.

1. De drager is {0, 1, 2, 3, ...}, waarbij de operator 0 het getal 0 voorstelt, 1 het getal 1, + de optelling en * de vermenigvuldiging.

2. De drager is de verzameling der reële getallen, 0 stelt nul voor, 1 stelt één voor, + de optelling en * de vermenigvuldiging.

3. De drager is termen(int) (dat wil zeggen het Herbrand-universum), 0 representeert de term '0', 1 representeert '1', de operator + representeert de concatenatie van de strings '+ (', eerste operand, ',' , tweede operand en ')'. Op dezelfde manier representeert *, als x en y de operanden zijn, '*(x, y)'.

4. De drager is {0, 1}, 0 representeert het getal 0, 1 het getal 1, + de optelling modulo 2 en * de vermenigvuldiging.

5. De drager is {0}, 0 en 1 representeren beide het getal 0, + representeert de optelling en * de vermenigvuldiging.

6. De drager is $\{a, b\}^*$ (dat wil zeggen de verzameling van alle strings die uit letters *a* en *b* bestaan), 0 representeert '*a*', 1 representeert '*b*', en + en * representeren beide de concatenatie.

Alle zes de boven beschreven algebra's zijn potentiële kandidaten voor het gegevenstype int, waarbij de eerste algebra de gebruikelijke betekenis belichaamt van de symbolen die we gebruiken. Het tweede voorbeeld heeft een veel grotere drager voor dezelfde signatuur. De derde algebra is de algebra die een parser of een symbolische evaluator zou kunnen gebruiken. De vierde is equivalent met een Booleaanse algebra, waarbij het toepasselijk zou zijn de symbolen te vervangen door *false*, *true*, *and* en *or*. De vijfde algebra is triviaal en de zesde is zo maar een algebra.

We willen de relatie onderzoeken tussen de drager van soort *s* van een Σ-algebra en de verzameling termen(*s*) voor soort *s*. Er bestaat een natuurlijke 1-op-1-correspondentie tussen termen(int) en de drager van algebra 3 hierboven, gewoon omdat ze dezelfde verzameling zijn. Algebra 3 is een voorbeeld van een speciale algebra, die onder verschillende namen bekend staat, namelijk de *vrije algebra voortgebracht door* Σ, een *initiële* Σ*-algebra* en de Σ*-woord-algebra*. Wat we dus eigenlijk willen onderzoeken is de relatie tussen algebra 3 en de andere hierboven opgesomde algebra's.

Relaties tussen algebra's worden bondig uitgedrukt in homomorfismen. We zullen het geval met één soort bekijken; de definitie kan gemakkelijk worden uitgebreid naar het geval met meer soorten. Een homomorfisme van Σ-algebra α naar Σ-algebra β met dragers *A* en *B* is een afbeelding $h: A \rightarrow B$, zodanig dat alle operaties (in Σ) bewaard blijven. Constanten blijven bewaard als, voor elke constante σ, geldt:

$$\mathrm{h}(\sigma_\alpha) = \sigma_\beta$$

Operaties die niet nul-air zijn blijven bewaard als voor elke operator σ en waarden $w_1, ..., w_n$ in *A* geldt:

$$h(\sigma_\alpha (w_1, ..., w_n)) = \sigma_\beta (h(w_1), ..., h(w_n))$$

Men kan zich de structuur van een algebra bepaald denken door de operatoren. Een homomorfisme bewaart de structuur van een algebra en laat expliciet zien

hoe de structuur van de ene algebra wordt afgebeeld op de structuur van de andere algebra. Een *isomorfisme* is een homomorfisme dat bijectief is (één-éénduidig en *op*). Isomorfe algebra's hebben een identieke structuur, maar de waarden van de dragers kunnen verschillende namen hebben. We zullen isomorfe algebra's vaak als gelijk beschouwen en uitspraken over isomorfe klassen van algebra's maken in plaats van over een specifieke algebra.

Σ-algebra's zijn belangrijk omdat er voor elke Σ-algebra α precies één homomorfisme bestaat van de woordalgebra naar α. Er bestaat een natuurlijke afbeelding $h_{\alpha,s}$: termen(s) $\rightarrow A_s$ voor elke soort s (we zullen de indices bij h weglaten als α en s vanzelfsprekend zijn). Dit unieke homomorfisme is

voor een constante σ :
$h(\sigma) = \sigma_\alpha$

voor een operator σ: $s_1 \times ... \times s_n \rightarrow s$
en termen $t_1, ..., t_n$ van de soorten $s_1, ..., s_n$:
$h(\sigma\ (t_1, ..., t_n)) = \sigma_\alpha\ (h(t_1), ..., h(t_n))$

Een waarde w van een Σ-algebra wordt bereikbaar genoemd als er in de woordalgebra een term t bestaat waarvoor geldt:

$h(t) = w$

waarin h het homomorfisme van de woordalgebra is. Een *onbereikbare* waarde is een waarde die niet kan worden uitgedrukt met behulp van de operatoren in Σ. Van algebra 2 zijn alleen de positieve gehele waarden bereikbaar; alle andere elementen van algebra 2 zijn onbereikbaar. Van de algebra's 1, 3, 4, 5 en 6 zijn alle elementen bereikbaar. Σ-algebra's met onbereikbare waarden zijn niet nuttig voor toepassing bij gegevenstypen, omdat de onbereikbare waarden voor de programmeur ontoegankelijk zijn. We zullen daarom Σ-algebra's met onbereikbare elementen uitsluiten (of althans de onbereikbare elementen van zo'n algebra negeren). Als we de onbereikbare elementen van algebra 2 verwijderen krijgen we algebra 1.

Twee termen x en y worden *equivalent* genoemd met betrekking tot een Σ-algebra α met drager A dan en slechts dan als $h(x) = h(y)$, waarin h het homomorfisme van de woordalgebra is. De equivalentie van deze twee termen impliceert dat de overeenkomstige waarde in A kan worden uitgedrukt als x of als y. De termen '1 + 0' en '0 + 1' zijn equivalent met betrekking tot de algebra's 1, 2,

4 en 5, maar niet met betrekking tot algebra 3 en 6. Merk op dat met betrekking tot de triviale algebra 5 alle termen equivalent zijn, maar dat in algebra 3 twee verschillende termen nooit equivalent zijn. Twee termen die met betrekking tot algebra 1 equivalent zijn, zijn ook equivalent met betrekking tot algebra 2.

Een equivalentie-relatie is reflexief, symmetrisch en transitief. Een *congruentie-relatie* is een equivalentierelatie met de eigenschap dat de relatie door (een bepaalde verzameling van) operatoren bewaard wordt, dat wil zeggen als x_i equivalent is met y_i (voor $1 \leq i \leq n$), dan is $\sigma\,(x_1, ..., x_n)$ equivalent met $\sigma\,(y_1, ..., y_n)$. Zij R een congruentie-relatie op een 1-soortige Σ-algebra α met drager A (R is congruent met betrekking tot de operatoren van Σ). Dan zullen we de congruentie-klasse waartoe het element x behoort aanduiden als

$$[x] = \{x' \mid R(x, x')\}$$

Twee elementen worden *congruent* genoemd als ze tot dezelfde congruentieklasse behoren. Een congruentierelatie kan worden beschouwd als een verzameling disjuncte deelverzamelingen waarvan A de vereniging is, namelijk $\{[x] \mid x \in A\}$, en operaties die als volgt worden gedefinieerd:

voor een constante σ :

$$\sigma_{\alpha/R} = [\sigma_\alpha]$$

voor een operator $\sigma : s_1 \times ... \times s_n \rightarrow s$ en waarden $x_1, ..., x_n$:

$$\sigma_{\alpha/R}\,([x_1], ..., [x_n]) = [\sigma_\alpha\,(x_1, ..., x_n)]$$

De eigenschap dat R een congruentierelatie is garandeert dat $\sigma_{\alpha/R}$ gedefinieerd is. Congruentierelaties en quotiënt-algebra's kunnen op een natuurlijke manier worden uitgebreid voor het veelsoortige geval.

De relatie 'x en y zijn equivalent met betrekking tot een Σ-algebra α' is een congruentierelatie op de verzameling van alle termen. De door deze relatie voortgebrachte quotiënt-algebra is isomorf met α. Als alternatief voor het zoeken naar een Σ-algebra kunnen we dus ook zoeken naar een congruentierelatie op de woordalgebra.

Quotiënt-algebra's zijn soms lastig te gebruiken omdat de elementen congruentieklassen zijn. Het is gemakkelijker een bepaald element van een klasse te gebruiken als representant van de gehele congruentieklasse. Een *canonieke vorm* is een deelverzameling C van termen(s), zodanig dat C niet twee elementen be-

vat die tot dezelfde congruentieklasse behoren en dat elke term t in termen(s) equivalent is met een term in C. Een *canonieke term-algebra* gebruikt een canonieke vorm als drager. De operaties worden gedefinieerd door het natuurlijke homomorfisme $h(x) = [x]$. De verzamelingen $\{0, 1\}$ en $\{1, 1 + 1\}$ zijn canonieke vormen voor algebra 4. De verzameling $\{0, 1, 1 + 1, 1 + (1 + 1), 1 + (1 + (1 + 1)), ...\}$ is een canonieke vorm voor algebra 1 en 2. De enige canonieke vorm voor algebra 3 is termen(int) en voor algebra 5 is elke verzameling van n elementen een canonieke vorm.

We kunnen nu de belangrijkste stelling van deze paragraaf formuleren: een gegevenstype van signatuur Σ is een isomorfisme-klasse van Σ-algebra's zonder onbereikbare waarden. Daarom kan een gegevenstype op de volgende equivalente manieren worden gerepresenteerd:

1. een Σ-algebra zonder onbereikbare waarden;
2. een congruentierelatie op de woordalgebra van Σ;
3. een quotiënt-algebra op de woordalgebra van Σ;
4. een canonieke term-algebra.

Een canonieke vorm identificeert niet altijd een uniek gegevenstype; in het volgende hoofdstuk zien we daarvan een voorbeeld. Canonieke vormen met de juiste operaties vormen een canonieke term-algebra die wel precies één gegevenstype representeert. Maar een gegevenstype kan worden gerepresenteerd door meer dan één canonieke term-algebra of Σ-algebra. Van de boven gedefinieerde algebra's representeren alleen nummer 2 en 3 gegevenstypen op een unieke manier, dat wil zeggen elk gegevenstype wordt door slechts één congruentierelatie (of quotiënt-algebra) gerepresenteerd.

Algebra's zijn geen volledige afspiegeling van de eigenschappen die we met gegevenstypen associëren. Enkele van de belangrijkste verschillen liggen bij fouten, zij-effecten en non-determinisme. Er bestaan twee gebruikelijke methoden om het probleem van fouten (zoals delen door nul of index buiten de grenzen) aan te pakken. De ene is het invoeren van fout-elementen in de algebra's; de andere is het toestaan van partiële functies als operaties. Beide methoden zijn geprobeerd; Goguen et al. (1978) bijvoorbeeld voegen fout-elementen aan elke soort toe en geven dan technieken voor het opstellen van axioma's. Het uitbreiden van operaties met partiële functies ziet er gemakkelijker uit en lijkt geschikter voor toepassingen in de informatica; deze methode is voorgesteld door Majster (1979) en door Broy en Wirsing (1982). Wirsing et al. (1983) beredeneren op overtuigende wijze dat specificaties eenvoudiger worden gemaakt

door partiële functies. Zij stellen : "Door de noodzaak om aan elke term een (gedefinieerd) object te koppelen – zelfs als een term volkomen onzinnig is – worden specificaties langer en de algebraïsche grondstructuur duisterder."

De algebraïsche aanpak gaat uit van waarde-gerichte gegevenstypen, waardoor zij-effecten worden uitgesloten. Niemand heeft serieus geprobeerd dit probleem op te lossen, omdat variabele-gerichte gegevenstypen meestal zonder veel problemen in de vorm van waarde-gerichte gegevenstypen kunnen worden gegoten. Maar die conversie kan een kleine aanpassing van de verzameling operatoren vergen. Een operator met meervoudige uitvoerwaarden wordt veranderd in een collectie operatoren, één voor elke uitvoerwaarde. Ook niet-deterministische operaties kunnen niet worden behandeld, aangezien een operatie bij gegeven operanden een vaste waarde aflevert. Beschouw bijvoorbeeld een verzamelingsoperatie *element* die een willekeurig gekozen element van een verzameling aflevert of die gewoon afhankelijk is van de implementatie. Zo'n operatie kan in de algebraïsche opzet niet worden gespecificeerd.

Op dit punt zou het op zijn plaats zijn voorbeelden van gegevenstypen te geven met gebruikmaking van de algebraïsche methode. Maar omdat bij de algebraïsche methode als natuurlijke manier voor het beschrijven van gegevenstypen ook algebraïsche specificaties behoren, stellen we de voorbeelden uit tot het volgende hoofdstuk, dat algebraïsche specificaties tot onderwerp heeft.

Opgaven

1. Beschouw de zes Σ-algebra's uit paragraaf 12.5. In welke algebra's zijn de volgende tweetallen termen equivalent?

   $1 + 0$ en $1 + 1 + 1$
   $(1 + 1) + 0$ en $1 + (1 + 0)$

2. Bewijs dat Σ-algebra 2 zonder de onbereikbare waarden isomorf is met Σ-algebra 1.

3. Geef enkele canonieke vormen voor algebra 6.

4. Hoe kan in de algebraïsche (of de domein-)methode een operatie worden vormgegeven die van de historie afhangt. Eén van de historie afhankelijke operatie is een operatie die de resulterende waarde berekent niet alleen op

grond van het operand, maar ook op grond van waarden die eerder aan de operatie zijn doorgegeven. Vele generatoren van pseudo-random getallen gebruiken bijvoorbeeld een statische lokale variabele die bij elke aanroep verandert en die wordt gebruikt voor het berekenen van de af te leveren waarde.

5. Gegeven een verzameling typen en polymorfe typen kan men een signatuur van typen construeren (misschien is *meta-signatuur* een goede naam). Elk niet-polymorf type is daarin een constante en elk polymorf type is een niet nul-aire operator. Beschouw bijvoorbeeld de typen int, bool, verzameling(*Z*) en afbeeldingen van *X* naar *Y*; de type-signatuur is dan:

int	: → type
bool	: → type
verzameling	: type → type
afbeelding	: type × type → type

Construeer een signatuur voor de typen van Pascal (of van een andere taal)

Literatuur

Oplossingen van recursieve domein-vergelijkingen zijn voor het eerst door Scott aangegeven in een aantal artikelen (1970, 1976) en gepresenteerd door Stoy (1977) en Tennent (1976). Het construeren van domeinen is vooral gebruikt voor het definiëren van de semantiek van programmeertalen. De algebraïsche methode in dit boek is gebaseerd op de methode met een initiële algebra, ontwikkeld door de ADJ-groep (Goguen en Thatcher, 1974; Goguen et al. 1975, 1978). Er zijn ook andere methoden ontwikkeld, zoals eindalgebra's, die door Wand (1979) en Kamin (1980) *final algebras* worden genoemd, en door Hornung en Raulefs (1980) *terminal algebras*.

13

Algebraïsche specificaties

In dit hoofdstuk introduceren we een specificatietechniek op basis van *algebraïsche axioma's*, of kortweg *axioma's*. De axioma's zullen congruentierelaties op de woordalgebra specificeren. Zoals in het vorige hoofdstuk is aangetoond, wordt een gegevenstype door een congruentierelatie op de woordalgebra gerepresenteerd.

13.1 Enkelvoudige axioma's

Een enkelvoudig axioma is een tweetal termen x en y van dezelfde soort. Het axioma wordt geschreven als

$$x = y$$

Om verwarring te vermijden gebruiken we het symbool '$\equiv$' om aan te geven dat twee termen identiek gelijk zijn (teken voor teken). Een congruentierelatie *voldoet* aan een verzameling R van axioma's als voor elk enkelvoudig axioma $x = y$ in R geldt dat x en y congruent zijn. Er kunnen vele congruentierelaties zijn die aan een verzameling axioma's voldoen. Eén van de mogelijke standpunten is dat een axiomaverzameling de gehele klasse van congruentierelaties die aan de axiomaverzameling voldoen, moet specificeren. Een ander standpunt is dat de axiomaverzameling alleen de kleinste van de congruentierelaties moet specificeren die aan de axiomaverzameling voldoen. In dit boek zullen we ons op het laatste standpunt stellen.

Een congruentierelatie A is kleiner dan congruentierelatie B als elke congruentieklasse van A bevat is in een congruentieklasse van B. Als we een relatie beschouwen als een verzameling paren, dan betekent 'A is kleiner dan B' gewoon dat A een deelverzameling is van B. De kleinste congruentierelatie die aan een verzameling R van axioma's voldoet wordt genoemd: de congruentierelatie die door R wordt *voortgebracht*. Aangezien de verzameling R van enkelvoudige axioma's ook kan worden gezien als een verzameling paren, kunnen we zeggen dat R de kleinste congruentierelatie voortbrengt waarin R bevat is. De kleinste congruentierelatie waarin R bevat is, is uniek en bestaat altijd. De notatie

$$x =_R y$$

betekent dat de termen x en y congruent zijn in de congruentierelatie die door R wordt voortgebracht. Zoals gewoonlijk zullen we de index achterwege laten als R uit de context kan worden bepaald. We noemen x en y ook wel equivalente termen. We zullen verder de notatie

$$[x] = \{y \mid x = y\}$$

gebruiken om de congruentieklasse aan te geven waartoe de term x behoort. De volgende inductieve definitie van $=$ is nuttig omdat die laat zien hoe kan worden aangetoond dat termen equivalent zijn met betrekking tot een axiomaverzameling R.

$x = y$ dan en slechts dan als een van de volgende uitspraken geldt:

Basis: $x = y$ (een enkelvoudig axioma in R)
Reflexief: $x \equiv y$ (x en y zijn dezelfde term)
Symmetrisch: $y = x$
Transitief: er bestaat een z, zodanig dat $x = z$ en $z = y$
Congruentie: er is een operator σ waarvoor geldt
$x \equiv \sigma(x_1, ..., x_n)$ en $y \equiv \sigma(y_1, ..., y_n)$ en
voor elke $0 \leq i \leq n$ geldt $x_i = y_i$

Beschouw de volgende signatuur met de constante 0 en de unaire operator S:

0: $\rightarrow$ N
S: N $\rightarrow$ N

en de volgende axiomaverzamelingen:

$R_0 = \{\}$
$R_1 = \{0 = 0\}$
$R_2 = \{0 = S0\}$
$R_3 = \{0 = SS0\}$
$R_4 = \{S0 = SS0\}$

De congruentieklassen die door de vijf axiomaverzamelingen worden voortgebracht, zijn:

R_0, R_1	{0}, {S0}, {SS0}, ...
R_2	{0, S0, SS0, ...}
R_3	{0, SS0, SSSS0, ...}, {S0, SSS0, SSSSS0, ...}
R_4	{0}, {S0, SS0, SSS0, ...}

R_0 en R_1 beschrijven de woordalgebra, omdat geen twee gelijke termen equivalent zijn. R_2 beschrijft de triviale algebra van één element; alle termen zijn equivalent. Op grond van de basis is $0 \simeq S0$, vervolgens geldt op grondt van congruentie $S0 \simeq SS0$ en op grond van transitiviteit $0 \simeq SS0$. Dit proces kan onbeperkt doorgaan ($0 \simeq S0 \simeq SS0 \simeq SSS0 \simeq ...$). R beschrijft een algebra zoals de Booleaanse algebra met twee elementen en de complement-operator, voorgesteld door S. Ook R_4 beschrijft de Booleaanse algebra met twee elementen, maar nu representeert S een constante functie. De verzameling {0,S0} is zowel voor R_3 als voor R_4 een canonieke vorm. Daaruit blijkt dat identieke canonieke vormen nog geen identieke congruentierelaties impliceren.

Het blijkt dat elk gegevenstype met een signatuur die uit slechts één constante en één unaire operator bestaat, met één enkelvoudig axioma kan worden gedefinieerd. Sommige gegevenstypen met ingewikkelder signatuur kunnen niet zo eenvoudig worden beschreven. De meeste nuttige gegevenstypen vergen een oneindig groot aantal enkelvoudige axioma's. Daarom voeren we een krachtigere vorm van axioma's in.

13.2 Axioma's met variabelen

Een *variabele* van soort s is een symbool dat verschilt van alle operatoren. Als v_l een variabele is van soort s_i(voor $1 \le i \le n$), dan is een *term met variabelen* een term waarin nul of meer deeltermen van soort s_i vervangen zijn door de variabele v_i.

De notatie

$$t\,(v_1, ..., v_n)$$

geeft een term aan met variabelen v_1 tot en met v_n; en

$$t\,(t_1, ..., t_n)$$

geeft dezelfde term aan, maar met elke variabele v_i vervangen door de term t_i (waarbij de soort van v_i gelijk is aan de soort van t_i).

Een *axioma* bestaat uit een tweetal termen met variabelen. Evenals bij enkelvoudige axioma's wordt het gelijkteken gebruikt om de twee termen van elkaar te scheiden. Een enkelvoudig axioma is een axioma met nul variabelen. Elk axioma kan worden geformuleerd met een mogelijkerwijs oneindig groot aantal enkelvoudige axioma's die worden verkregen door substitutie van variabelen door van alle mogelijke termen van de juiste soort. Zo betekent bijvoorbeeld het axioma

$$r\,(v_1, ..., v_n) = s\,(v_1, ..., v_n)$$

hetzelfde als de axiomaverzameling

$$\{r\,(t_1, ..., t_n) = s(t_1, ..., t_n) \mid t_i \in \text{termen(soort van } v_i)\}$$

We gebruiken gewoonlijk kleine letters (en af en toe identifiers van kleine letters) voor variabelen. De soort van de variabele kan meestal uit de context worden bepaald. Om er gemakkelijker naar te kunnen verwijzen denken we de axioma's in een axiomaverzameling geordend en vanaf 1 genummerd in de volgorde waarin ze in de axiomaverzameling voorkomen. Beschouw de signatuur

$0{:} \to \mathrm{N}$
$\mathrm{S}{:}\ \mathrm{N} \to \mathrm{N}$
$+{:}\ \mathrm{N} \times \mathrm{N} \to \mathrm{N}$

en de axiomaverzamelingen

$R_5 = \{\ x + 0 = x, x + Sy = \mathrm{S}(x + y)\ \}$
$R_6 = \{\ 0 + x = x, Sy + x = \mathrm{S}(y + x)\ \}$

$R_7 = \{\ 0 + x\ = x, x\ + 0{=}x, Sx\ +\ Sy = x{+}y\ \}$
$R_8 = \{\ SSx = x, x + 0 = 0, x + S0 = x\ \}$

R_5 en R_6 specificeren hetzelfde gegevenstype, namelijk de integers – waarbij het getal nul wordt voorgesteld door de operator 0, de operatie 'opvolger' door S en optelling door +. De operatie 'opvolger' telt bij een getal één op. Om in te zien dat SS0 + S0 = SSS0 merken we eerst op dat we, door in axioma 2 van R_5 SS0 voor x en 0 voor y te substitueren, het enkelvoudige axioma SS krijgen; door in het eerste axioma SS0 voor x te substitueren krijgen we het axioma S-S0 + 0 = SS0. Met behulp van de congruentie-eigenschap kunnen we de juiste substitutie uitvoeren. We kunnen het bewijs als volgt samenvatten:

SS0 + S0	= S(SS0 + 0)	via axioma 2
	= S(SS0)	via axioma 1
	≡ SSS0	

Voor axiomaverzameling R_6 wordt het bewijs:

SS0 + S0	= S(S0 + S0)	via axioma 2
	= SS(0 + S0)	via axioma 2
	= SS(S0)	via axioma 1
	≡ SSS0	

De praktijk van het bewijzen van equivalentie van termen door termen met behulp van axioma's te herschrijven of te reduceren heeft geleid tot andere woorden voor axioma's, zoals *herschrijfregel, reductieregel* en *vergelijking*.

De algebraïsche structuur van de gehele getallen brengt ons voor R_5 en R_6 tot de voor de hand liggende canonieke vorm C={0, S0, SS0, ...}. Om te bewijzen dat C een canonieke vorm is, bewijzen we eerst dat elke term equivalent is met één van de canonieke termen en vervolgens dat er geen twee verschillende canonieke termen zijn die equivalent zijn. Door inductie op de axioma's kunnen we aantonen dat elke term met een operator + kan worden herschreven of gereduceerd tot een term zonder operator +. We kunnen ook aantonen dat de axioma's het aantal operatoren S in een term constant houden en dat, aangezien elk tweetal verschillende canonieke termen een verschillend aantal operatoren S bevat, twee canonieke termen niet equivalent kunnen zijn.

R_7 beschrijft ook de gehele getallen, maar de operator + representeert hier het verschil in plaats van de som. Dat C een canonieke vorm is voor R_7 is iets moei-

lijker aan te tonen, omdat het derde axioma het aantal operatoren S niet constant houdt. R_8 beschrijft een Booleaanse algebra van twee elementen, waarbij 0 staat voor false, S voor het complement en + voor de Booleaanse operatie *and*. Men kan R_8 ook beschouwen als een beschrijving van de Booleaanse algebra van twee elementen waarbij 0 staat voor true, S voor het complement en + voor de Booleaanse operatie *or*. Deze twee Booleaanse algebra's zijn isomorf en zijn daarom beide een plausibele interpretatie van R_8.

13.3 Verborgen operatoren

We stellen nu de volgende voor de hand liggende vraag: is een eindig aantal axioma's machtig genoeg om elk gegevenstype te beschrijven? Majster was één der eersten die dit probleem hebben onderzocht. Zij gebruikte het voorbeeld van een doorloopbare stack om tot de conclusie te komen dat eindige axiomaverzamelingen niet machtig genoeg zijn. Dit resultaat bracht een langdurige correspondentie in *SIGPLAN Notices* op gang. De speelgoedstack is een vereenvoudigde versie van de doorloopbare stack (Thatcher et al., 1979). Hier volgt een specificatie van de speelgoedstack met een oneindige axiomaverzameling.

0: $\rightarrow$ T *lege stack*
E: $\rightarrow$ T *fout*
P: T $\rightarrow$ T *push*
D: T $\rightarrow$ T *neer – kijk naar de volgende waarde in de stack*

$$R_9 = \{DE = E, PE = E, PDx = E\} \cup \{D^n P^k 0 = E \mid n > k\}$$

waarin D^n een verkorte notatie is voor DDDD... (*n* keer). De verzameling van congruentieklassen die door R_9 wordt voortgebracht = $\{ \{D^n P^k 0\} \mid n \leq k\} \cup \{ \{E, D0, DE, PE, ...\} \}$; daarom is $\{D^n P^k 0 \mid n \leq k\} \cup \{E\}$ een natuurlijke canonieke vorm.

Er bestaat geen eindige axiomaverzameling voor de speelgoedstack; zie Thatcher en anderen (1979) voor een bewijs. Dit struikelblok kan worden verwijderd met behulp van verborgen operatoren.

Een *verborgen operator* is een nieuwe operator die uitsluitend voor specificatiedoeleinden aan de signatuur wordt toegevoegd. Voor de gebruiker van het type is deze operator verborgen of ontoegankelijk. Voor praktische doeleinden

blijft de signatuur van het gegevenstype onveranderd (behalve voor het praktische doel van specificatie). Met behulp van verborgen operatoren kan men elk berekenbaar gegevenstype met een eindig aantal axioma's definiëren (Guttag, 1980). De speelgoedstack kan bijvoorbeeld worden gespecificeerd met behulp van de verborgen operator die we H noemen. In de volgende en latere specificaties zullen verborgen operatoren met een sterretje worden gemarkeerd.

	0:	$\to$ T
	E:	$\to$ T
	P:	T $\to$ T
	D:	T $\to$ T
*	H:	T $\to$ T

$$R_{10} = \{DE = E, PE = E, HE = E, D0 = E,$$
$$PDx = E, PHx = E,$$
$$DHx = HDx,$$
$$DP0 = H0,$$
$$DPPx = HPx\}$$

Bij het gebruik van verborgen operatoren wordt het noodzakelijk enkele definities te verduidelijken. Als een signatuur Σ, een omvattende verzameling $\Sigma' = \Sigma \cup$ {de verborgen operatoren} en een axiomaverzameling R voor Σ' gegeven zijn, dan zal de door R voortgebrachte congruentierelatie beperkt blijven tot de verzameling congruentieklassen

$$A' = \{[t] \mid t \in \text{termen}(\Sigma', s)\}$$

in plaats van

$$A = \{ [t] \mid t \in \text{termen}(\Sigma, s)$$

De congruentieklasse $[t]$ blijft gedefinieerd als:

$$[t] = \{t' \mid t' = t)$$

Er is misschien geen verschil tussen A en A', maar als er wel verschil is, betekent dat, dat er in A' een congruentieklasse bestaat waarvan elke term minstens één verborgen operator heeft. Daardoor is deze congruentieklasse *onbereikbaar* en wordt deze niet als een waarde van het gegevenstype beschouwd.

13.4 Conditionele axioma's

In sommige formuleringen van algebraïsche systemen wordt nog een ander soort axioma's gebruikt. Twee gebruikelijke vormen van conditionele axioma's zijn:

$$p = \text{if } q \text{ then } r \text{ else } s$$
$$(q) => p = r$$

waarin p, r en s termen van dezelfde soort zijn en q een Booleaanse term. De constructie *if then else* is in deze formulering geen operator, maar betekent dat p equivalent is met r als q waar is en dat p anders equivalent is met s. De betekenis van de tweede vorm van conditionele axioma's is, dat p equivalent is met r als q waar is. Beide bovenstaande conditionele formules kunnen met een operator *ifthenelse* ook gemakkelijk worden geschreven. Als de operator *ifthenelse* niet beschikbaar is, dan kan deze als verborgen operator worden toegevoegd. De tweede formule kan dan worden geschreven als

$$p = \text{ifthenelse}(q, r, p)$$

of in een meer leesbare vorm:

$$p = \text{if } q \text{ then } r \text{ else } p$$

Waar nodig zullen we er voor elke soort Z van uitgaan dat de mogelijk verborgen operator ifthenelse aanwezig is. De axioma's voor ifthenelse zijn:

$$R_{11} = \{\text{if true then } x \text{ else } y = x,$$
$$\text{if false then } x \text{ else } y = y\}$$

13.5 Extensies en verrijkingen

Een algebraïsche specificatie omvat vijf delen:

1. de naam van de specificatie;
2. een lijst soorten;
3. een signatuur;
4. een lijst van variabelen met hun soort;
5. een lijst axioma's;

Een specificatie beschrijft een verzameling gegevenstypen. De voor elke soort door de axioma's voortgebrachte congruentierelaties specificeren een gegevenstype. Een specificatie van het Booleaanse gegevenstype vormt een illustratie van het formaat dat we hebben gekozen.

Naam:	EersteVoorbeeld	
Soorten:	Bool	
Operatoren:		
	true:	$\rightarrow$ Bool
	false:	$\rightarrow$ Bool
	ifthenelse:	Bool × Bool × Bool $\rightarrow$ Bool
	not:	Bool $\rightarrow$ Bool
	and:	Bool × Bool $\rightarrow$ Bool
	or:	Bool × Bool $\rightarrow$ Bool
Variabelen:		
	x, y:	Bool

Axioma's:

if true then x else $y = x$
if false then x else y = y
not x = if x then false else true
x and y = if x then y else false
x or y = if x then true else y

In de context van programmeren worden gegevenstypen meestal opgebouwd met inachtneming van hun relaties tot andere gegevenstypen. Men neemt in het algemeen aan dat de gegevenstypen Boolean en integer aanwezig zijn en dat er predikaten en numerieke operaties op bestaan. Bij de opbouw van een gegevenstype string bijvoorbeeld hebben de vergelijkingsoperatoren een Booleaans type nodig, en de operator *lengte* zal een integer als resultaat hebben. Omdat het onhandig is elke keer de gehele specificatie van het type Boolean te kopiëren als die nodig is voor een andere specificatie, wordt het concept *extensie* ingevoerd.

Een *extensie* van een eerder gedefinieerde specificatie T is een nieuwe specificatie T' die, naast alle eerder gedefinieerde soorten, operatoren en axioma's van T, ook nieuwe soorten, operatoren en axioma's bevat, met de eigenschap dat alle typen van T in T' onveranderd blijven. Deze eigenschap wordt *bescherming* of *protectie* genoemd en wordt later precies gedefinieerd. Een *verrijking* is een

extensie waarin geen nieuwe soorten worden toegevoegd. Voor extensies en verrijkingen wordt hetzelfde formaat gebruikt als voor algebraïsche specificaties, maar ze verwijzen naar de naam van de oude specificatie waarop ze voortbouwen en ze geven een expliciete lijst van de nieuwe soorten, operatoren en axioma's. Het formaat van zo'n specificatie is:

Naam:	*naam* (extensie van | verrijking van) *lijst van namen*
Soorten:	*lijst van nieuwe soorten*
Operatoren:	*lijst van nieuwe operatoren met hun ariteit*
Variabelen:	*lijst van variabelen met hun soort*
Axioma's:	*lijst van nieuwe axioma's*

De lijst van namen die volgt op de sleutelwoorden *extensie van* of *verrijking van* specificeert de namen van specificaties die alle oude soorten, operatoren en axioma's bevatten. De afdeling soorten kom in verrijkingsspecificaties niet voor. De afdeling variabelen zal meestal achterwege blijven als de soort van de variabelen gemakkelijk uit de context kan worden bepaald. Elke specificatie kan worden beschouwd als een extensie van de lege specificatie. Dit is de eenvoudigste Booleaanse specificatie met slechts twee operatoren:

Naam:	EenvoudigeBoolean	
Soorten:	Bool	
Operatoren:		
	true:	$\rightarrow$ Bool
	false:	$\rightarrow$ Bool
Axioma's:	geen	

Het is duidelijk dat de unieke canonieke vorm {true,false} is. Nu volgen eenvoudige voorbeelden van een verrijking en een extensie van Eenvoudige Boolean.

Naam:	EenvoudigeBoolean
Naam:	VerrijkteBoolean verrijking van EenvoudigeBoolean
Operatoren:	ifthenelse: Bool × Bool × Bool $\rightarrow$ Bool
Axioma's:	if true then *x* else y = *x*
	if false then *x* else y = y

Naam: EenvoudigeInt extensie van Boolean
Soorten: int
Operatoren:

0:		$\rightarrow$ int
S:		int $\rightarrow$ int
$\leq$:		int $\times$ int $\rightarrow$ Bool

Axioma's:

$0 \leq x$	= true
$Sx \leq 0$	= false
$Sx \leq Sy$	$= x \leq y$

Omdat er geen axioma's zijn die termen van de soort int herschrijven, is de canonieke vorm voor int termen(int) = {0, S0, SS0, ...}.

13.6 Protectie, volledigheid en consistentie

Van de termen *volledigheid, consistentie* en *protectie (of* bescherming) is een precieze definitie nodig. We willen protectie geven aan oude typen om ervoor te zorgen dat ze door extensies niet worden veranderd. Zij T een specificatie van een verzameling gegevenstypen met signatuur Σ en axiomaverzameling A', en zij T' een andere specificatie met signatuur Σ' en axiomaverzameling A', waarvan A' een deelverzameling is. Bij een soort s zij R de congruentierelatie voortgebracht door A en R' de congruentierelatie voortgebracht door A'. Als elke congruentieklasse in R' een term bevat uit een van de congruentieklassen van R, dan wordt T' *s-volledig met betrekking tot* T genoemd. Gevoelsmatig komt dat erop neer dat er geen nieuwe waarden voor s zijn ontstaan bij de overgang van T naar T'. Als voor elk paar x, y van niet-congruente termen in R geldt dat x en y niet congruent zijn in R', dan wordt T' *s-consistent met betrekking tot* T genoemd. Gevoelsmatig betekent dit, dat verschillende waarden van s in T, in T' nog steeds verschillend zijn. Als alle soorten s in T s-volledig zijn, wordt T' *volledig met betrekking tot* T genoemd. (Wand, 1979, noemt dit Λ-*getrouw*.) Als T' zowel consistent als volledig is met betrekking tot T, dan is T *beschermd* in T' en is T' een legale *extensie* van T.

Volledigheid garandeert dat er door extensie geen nieuwe waarden (van oude typen) worden gecreëerd, en consistentie garandeert dat afzonderlijke waarden door de extensie niet tot één waarde worden samengevoegd. Onvolledigheid houdt in dat we meer (of sterkere) axioma's nodig hebben, inconsistentie houdt in dat we minder (of zwakkere) axioma's nodig hebben. Eén van de manieren

om aan te tonen dat een soort s zowel volledig als consistent is, is te bewijzen dat s in T' en in T dezelfde canonieke vorm heeft.

Om aan te tonen dat EenvoudigeBoolean inderdaad beschermd is in VerrijkteBoolean, moeten we aantonen dat {true, false} een canonieke vorm voor de soort Bool is in VerrijkteBoolean. Door inductie blijkt dat elke term waarin de operator *ifthenelse* voorkomt herschreven kan worden zonder *ifthenelse*, omdat het eerste operand tot *true* of *false* kan worden herleid. Dan moet nog worden bewezen dat *true* niet equivalent is met *false*. Dat is meestal het moeilijkste deel van dit soort bewijs.

13.7 Afgeleide operatoren

Een *afgeleide operator* is een operator die kan worden gedefinieerd als een term met variabelen. De Booleaanse operaties *complement, and* en *or* bijvoorbeeld kunnen worden gedefinieerd in termen van de operator *ifthenelse*. Een afgeleide operator σ kan worden gedefinieerd met precies één niet-recursief axioma van de vorm

$$\sigma\,(x_1, ..., x_n) = t\,(x_1, ..., x_n)$$

Zulke afgeleide operatoren veroorzaken nooit onvolledigheid of inconsistentie. Afgeleide operatoren kunnen worden beschouwd als afkortingen van termen en daarom veranderen ze de canonieke vorm niet. Hier volgt een voorbeeld van een verrijkt type waarin alleen afgeleide operatoren worden gebruikt.

Naam:	Boolean verrijking van VerrijkteBoolean	
Operatoren:	not:	Bool → Bool
	and:	Bool × Bool → Bool
	or:	Bool × Bool → Bool
Axioma's:	not x	= if x then false else true
	x and y	= if x then y else false
	x or y	= if x then true else y

De integers met de gebruikelijke aritmetische operaties vormen een ander nuttig gegevenstype:

Naam: Integer verrijking van EenvoudigeInt, Boolean

Operatoren:

+: Int × Int → Int
*: Int × Int → Int
**: Int × Int → Int
=: Int × Int → Bool
≠: Int × Int → Bool
≥: Int × Int → Bool
<: Int × Int → Bool
>: Int × Int → Bool
1: → Int
2: → Int

Axioma's:

$0 + x = x$ $Sy + x = S(x+y)$
$0 * x = 0$ $Sy * x = x + y * x$
$X ** 0 = 1$ $x ** Sy = x * x ** y$

$(x = y)$ $= x \leq y$ and $y \leq x$
$(x \neq y)$ $=$ not $x = y$
$(x \geq y)$ $= y \leq x$ and $x \neq y$
$(x > y)$ $=$ not $x \leq y$
$(x < y)$ $=$ not $x \geq y$

$1 = S0$
$2 = S1$

Merk op dat alle bovenstaande vergelijkingsoperatoren en constanten afgeleide operatoren zijn en dat elke aritmetische operator recursief is gedefinieerd op de canonieke vorm die uit EenvoudigeInt is bepaald.

De voorbeelden met integer en Boolean zijn in stadia opgebouwd. De specificatie van Boolean is equivalent met de specificatie van EersteVoorbeeld. Het doel van de stadia is de eenvoudige bewijsbaarheid van de correctheid van de specificatie. In het eenvoudige eerste stadium van Bool en Int is de canonieke vorm gemakkelijk te bepalen en is de juistheid ervan gemakkelijk vast te stellen aan de hand van ons intuïtieve begrip van Booleaanse en integer waarden. In dit eerste stadium zoeken we naar de eenvoudigste verzameling operaties die alle waarden van een gegevenstype voortbrengen. In EenvoudigeInt hadden we de definitie van ≤ nog kunnen uitstellen tot het stadium integer. De operaties uit de

eerste stadia worden meestal *constructor*-operaties genoemd omdat ze alle waarden van een gegevenstype construeren. Ze zijn nuttig voor het bepalen van geschikte canonieke vormen.

Als de constructor-operaties eenmaal zijn bepaald, kunnen de overige operaties op een ongecompliceerde manier worden gedefinieerd. Afgeleide operaties zijn het gemakkelijkst. We kunnen de volledigheid van andere operaties op het oog controleren door na te gaan of elke term die de operator bevat, herleid kan worden tot een term zonder de operator. We mogen ervan uitgaan dat de operanden van de operator in canonieke vorm staan (omdat het bewijs meestal inductief verloopt). Zoals gewoonlijk is het bewijs van de consistentie moeilijker.

13.8 Geparametriseerde gegevenstypen en verzameling(Z)

We zullen nu de specificatie van geparametriseerde typen bekijken aan de hand van het verzamelingenvoorbeeld. Met de eerder besproken methoden is de definitie van verzameling van Int eenvoudig. We willen echter *verzameling van* Z specificeren voor elk type Z. Z heet een type-parameter en we mogen ervan uitgaan dat er bij het type bepaalde operaties horen. In het volgende voorbeeld eisen we dat Z een gelijkheidsoperator heeft, zodat *verzameling van* Z alleen is gedefinieerd voor typen met een gelijkheidsoperator. Een type-parameter moet beschermd zijn, net als alle andere eerder gedefinieerde operatoren. De afdeling soorten van de specificatie moet uitgebreid worden met geparametriseerde soorten en de operaties daarop.

Naam: verzameling extensie van Boolean
Soorten: verzameling(Z met gelijk: Z × Z → Bool)
Operatoren:

lege-verzameling:	→ verzameling(Z)
voeg-toe:	Z × verzameling(Z) → verzameling(Z)
verwijder:	Z × verzameling(Z) → verzameling(Z)
lid:	Z × verzameling(Z) → Bool
vereniging:	verzameling(Z) × verzameling(Z) → verzameling(Z)
doorsnee:	verzameling(Z) × verzameling(Z) → verzameling(Z)
deelverzameling:	verzameling(Z) × verzameling(Z) → Bool
verzamelingsgelijk:	verzameling(Z) × verzameling(Z) → Bool

Axioma's:

1. voeg_toe(x, voeg_toe(y, v)) = if gelijk(x, y)
then voeg_toe(x, v)
else voeg_toe(y, voegtoe(x, v))

2. verwijder(x, lege_verzameling) = lege_verzameling

3. verwijder(x, voeg_toe(y, v)) = if gelijk(x, y)
then verwijder(x, v)
else voeg_toe(y, verwijder(x, v))

4. lid(x, lege_verzameling) = false

5. lid(x, voeg_toe(y, v)) = if gelijk(x, y)
then true
else lid(x, v)

6. vereniging(v, lege_verzameling) = v

7. vereniging(v, voeg_toe(x, w)) = voeg_toe(x, vereniging(v, w))

8. doorsnee(v, lege_verzameling) = lege_verzameling

9. doorsnee(v, voeg_toe(x, w)) = if lid(x, v)
then voeg_toe(x, doorsnee(v, w))
else doorsnee(v, w)

10. deelverzameling(lege_verzameling, v) = true

11. deelverzameling(voeg_toe(x, v), w) = if lid(x, v)
then deelverzameling(v, w)
else false

12. verzamelingsgelijk(v, w) =
deelverzameling(v, w) and deelverzameling(w, v)

In bovenstaand voorbeeld worden twee operatoren *ifthenelse* gebruikt. De ene is de operator die in VerrijkteBoolean is gedefinieerd en de andere is

ifthenelse: Bool × verzameling(Z) × verzameling(Z) → verzameling(Z)

Zoals gezegd aan het eind van de vorige paragraaf zullen we operatoren *ifthen else* gebruiken waar dat handig uitkomt. Het gebruik ervan impliceert dan de twee betreffende axioma's.

De constructors van het gegevenstype verzameling zijn *lege_verzameling* en *voeg_toe*. Alle andere operatoren (behalve de afgeleide operator *verzamelings-gelijkheid*) hebben twee axioma's, één voor het geval van de lege verzameling en de andere voor het geval van de operator voeg_toe. Zo kan elke term van de soort verzameling(Z) herleid worden tot een term met uitsluitend de operatoren *voeg_toe* en *lege_verzameling*.

Het beschrijven van een canonieke vorm is voor verzameling(Z) moeilijker dan voor Bool en Int. De verzameling

{ voeg_toe(z_1, ...voeg_toe(z_n, lege_verzameling)...) | $n \geq 0$}

is geen canonieke vorm omdat bijvoorbeeld

voeg_toe(x, voeg_toe(lege_verzameling)) =
voeg_toe(x, lege_verzameling)

Zelfs de verzameling

{ voeg_toe(z_1, ...voeg_toe(z_n, lege_verzameling)...) |
$n \geq 0$ en voor alle *i, j met* i ≠ j : niet gelijk(z_i, z_j)}

is geen canonieke vorm, aangezien

voeg_toe(x, voeg_toe(y, lege_verzameling)) =
voeg_toe(y, voeg_toe(x, lege_verzameling))

Als de elementen van Z geordend kunnen worden, dan is de verzameling

{ voeg_toe(z_1, ...voeg_toe(z_n, lege_verzameling)...) |
$n \geq 0$ en $z_1 < ... < z_n$}

een canonieke vorm voor verzameling(Z). Een ingewikkelde canonieke vorm maakt het controleren van de volledigheid en de consistentie van elke operator

moeilijk. Stel dat het then-deel van axioma 3 'then *v*' zou luiden in plaats van 'then verwijder(*x*, *v*)'. Op het eerste gezicht lijkt dit misschien een onschuldige verandering, maar de specificatie zou hierdoor Bool-inconsistent worden.

true
≡ lid(*x*, voeg_toe(*x*, lege_verzameling)) via axioma 5
≡ lid(*x*, verwijder(*x*, voeg_toe(*x*, voeg_toe(*x*, lege_verzameling))))
via axioma 3*
≡ lid(*x*, verwijder(*x*, voeg_toe(*x*, lege_verzameling)))
via axioma 1
≡ lid(*x*, lege_verzameling) via axioma 3*
≡ false via axioma 4

13.9 Stack(*Z*) en fouten

Dit voorbeeld is een eerste poging om een stack van *Z* te beschrijven. Het voorbeeld is niet *Z*-volledig en niet Bool-veilig.

Naam: OnvoltooideStack extensie van Boolean
Soorten: stack(Z)
Operatoren:
lege_stack: → stack(Z)
push: Z × stack(Z) → stack(Z)
pop: stack(Z) → stack(Z)
top: stack(Z) → Z
leeg?: stack(Z) → Bool
Axioma's:
1. pop(push(*z*, *s*)) = *s*
2. top(push(*z*, s)) = *z*
3. leeg?(lege_stack) = true
4. leeg?(push(*z*, *s*)) = false

Laten we, voordat we OnvoltooideStack analyseren, eens bedenken wat een canonieke vorm zou kunnen zijn, uitgaande van ons intuïtieve beeld van een stack. Elke stack kan worden gecreëerd met behulp van de operatoren *lege_stack* en *push*.

* Via de gewijzigde versie van axioma 3:
verwijder(*x*, voeg_toe(*y*,*v*) = if gelijk(*x*,*y*) then *v* else voeg_toe(*y*, verwijder(*x*,*v*)).

Daarom zou

$$C = \{ \text{push}(z_1, \ldots\text{push}(z_n, \text{lege_stack})\ldots) \mid n \geq 0 \}$$

een redelijke canonieke vorm zijn. Maar expressies met de operator *pop* kunnen niet altijd worden herleid tot expressies waarin alleen operatoren *lege_stack* en *push* voorkomen, met name de expressie pop(lege_stack). Daarom is

$$C \cup \{ \text{pop}(\ldots\text{pop}(\text{lege_stack})\ldots) \mid \text{voor 1 of meer keer pop} \}$$

een canonieke vorm voor stack(Z) van OnvoltooideStack. Op deze manier houden we een onbehaaglijk gevoel over stacks. Er zijn verschillende dingen die we kunnen doen om de situatie te verbeteren. Ten eerste kunnen we pop(lege_stack) als fout bestempelen en deze fout toevoegen aan onze intuïtieve canonieke vorm, zodat we krijgen:

$$C' = C \cup \{\text{stack-fout}\}$$

Deze redelijke en logische oplossing leidt tot het nieuwe axioma:

5. pop(lege_stack) = stack-fout

Maar nu komen er complicaties, omdat we moeten beslissen wat er moet worden afgeleverd door push(*z*, stack-fout), top(stack-fout) en leeg?(stack-fout). Het ziet ernaar uit dat we ook een foutwaarde moeten toevoegen aan de soorten Bool en *Z* en de volgende nieuwe axioma's moeten toevoegen:

6. top(stack-fout) = *z*-fout
7. pop(stack-fout) = stack-fout
8. leeg?(stack-fout) = Bool-fout
9. push(*z*, stack-fout) = stack-fout

Maar helaas zijn deze axioma's niet Bool-consistent en niet *Z*-consistent!

Bool-fout	= leeg?(fout)	via axioma 8
	= leeg?(push(*z*, fout))	via axioma 9
	= false	via axioma 4
z	= top(push(*z*, stack-fout))	via axioma 2
	= top(stack-fout)	via axioma 9

$= \text{top}(\text{push}(x, \text{stack-fout}))$ via axioma 9
$= x$ via axioma 2

Dit zijn enkele van de problemen die men tegenkomt bij het behandelen van fouten. Goguen et al. (1978) geven oplossingen voor deze problemen en presenteren methoden voor het integreren van fouten in algebraïsche specificaties. Een andere manier om fouten te behandelen in specificaties is het generaliseren van het begrip operatie naar partiële functie (Guttag et al., 1978; Majster, 1979; Kamin en Archer, 1984). Dan kunnen fouten in specificaties worden opgenomen zonder de problemen die we hierboven zijn tegengekomen.

Een andere, minder prettige aanpak van het probleem is het ontwerpen van gegevenstypen zonder fouten. Daartoe moeten we definiëren wat het effect van de operaties *pop* en *top* is op een lege stack. We kunnen pop(lege_stack) de lege stack laten afleveren, maar wat doen we met top(lege_stack)? Om de waarde daarvan te laten definiëren maken we van lege_stack een unaire operator die een waarde uit Z specificeert die als foutwaarde moet worden gebruikt. Deze afwijkende stack kan nu gemakkelijk als volgt worden gespecificeerd:

Naam:	AfwijkendeStack extensie van Boolean	
Soorten:	stack(Z)	
Operatoren:		
	lege_stack:	$Z \to \text{stack}(Z)$
	push:	$Z \times \text{stack}(Z) \to \text{stack}(Z)$
	pop:	$\text{stack}(Z) \to \text{stack}(Z)$
	top:	$\text{stack}(Z) \to Z$
	leeg?:	$\text{stack}(Z) \to \text{Bool}$

Axioma's:

1. pop(lege_stack(z)) = lege_stack(z)
2. pop(push(z, s)) = s
3. top(lege_stack(z)) = z
4. top(push(z, s)) = s
5. leeg?(lege_stack(z)) = true
6. leeg?(push(z, s)) = false

13.10 De specificatie van de lambdacalculus

Als laatste voorbeeld in deze paragraaf presenteren we een algebraïsche specificatie van de lambdacalculus. Dit is een extra goed voorbeeld omdat de lamb-

dacalculus een geliefde taal is voor het bestuderen van type-kwesties en ook omdat het de macht van de algebraïsche specificatiemethode laat zien. De grammatica voor de lambdacalculus is eenvoudig:

expressie → identifier
expressie → (expressie expressie)
expressie → (λ identifier . expressie)

In de lambdacalculus worden expressies opgebouwd uit identifiers, applicatie van een functie op een argument en abstractie. De lambdacalculus kent drie conversie-regels:

α	Als w niet vrij is in M, dan $(\lambda v.M) = (\lambda w.\text{sub}(M,v,w))$
β	$((\lambda x.M)N) = \text{sub}(M,x,N))$
η	Als x niet vrij is in M, dan $(\lambda x.(M\ x)) = M$

waarin sub(a,b,c) betekent: substitueer c op alle plaatsen waar b vrij voorkomt in a. De algebraïsche specificatie specificeert eerst het domein van identifiers (vergelijkbaar met de definitie van integers) en voert de drie operatoren *var*, *app* en *abs* in voor het opbouwen van expressies. De twee verborgen operatoren *sub* en *nietvrij* representeren substitutie en gebonden identifiers. De algebraïsche specificatie van de lambdacalculus is als volgt:

Naam: Lambdacalculus extensie van Boolean
Soorten: Exp, Id
Operatoren:

	eerste_id:	→ id
	volgende_id:	Id → Id
	gelijk:	Id × Id → Bool
	var:	Id → Exp
	app:	Exp × Exp → Exp
	abs:	Id × Exp → Exp
*	sub:	Exp × Id × Exp → Exp
*	nietvrij:	Id × Exp → Bool

Axioma's:

abs(v, M) = if nietvrij(w, M) – α-conversie
then abs(w, sub(M, v, var(w)))
else abs(v, M)

app(abs(x, M), N) = sub(M, x, N) – β-conversie

abs(x, app(M, var(x))) = if nietvrij(x, M) – η-conversie
then M
else abs(x, app(M, var(x)))
sub(var(y), x, E) = if gelijk(x, y) then E else var(y)
sub(app(M, N), x, E) = app(sub(M, x, E), sub(N, x, E))
sub(abs(y, M), x, E) = if gelijk(x, y)
then abs(y, M)
else if nietvrij(y, E)
then abs(y, sub(M, x, E)
else sub(abs(y,M), x, E)
nietvrij(x, var(y)) = not gelijk(x, y)
nietvrij(x, app(M, N)) = nietvrij(x, M) and nietvrij(x, N)
nietvrij(x, abs(y, M)) = gelijk(x, y) or nietvrij(x, M)

gelijk(x, x) = true
gelijk(eerste_id, volgende_id(x)) = false
gelijk(volgende_id(x), eerste_id) = false
gelijk(volgende_id(x), volgende_id(y)) = gelijk(x, y)

13.11 Begin- en eindalgebra's

De in dit en het vorige hoofdstuk gepresenteerde benadering van gegevenstypen is gebaseerd op initiële algebra's of begin-algebra's, ontwikkeld door de ADJ-groep (Goguen, Thatcher, Wagner en Wright). Een algebra A is *initieel* in een klasse van algebra's als er voor elke algebra X in de klasse een uniek homomorfisme van A naar X bestaat. Voor elke signatuur Σ is de Σ-woordalgebra initieel voor alle algebra's met Σ als signatuur. Dat betekent dat er een unieke afbeelding bestaat van termen naar de waarden van elke Σ-algebra.

De begin-algebra is een krachtig concept, want het houdt in dat men slechts zo'n algebra behoeft aan te wijzen om de afbeelding van termen naar elementen van de algebra gratis te krijgen. Bij het werken met algebraïsche specificaties geeft de kleinste congruentierelatie voortgebracht door een axiomaverzameling een algebra die initieel is onder alle algebra's die aan de axioma's voldoen. We krijgen deze initiële algebra door aan te nemen dat termen verschillende waarden representeren tenzij met de axioma's kan worden bewezen dat die termen equivalent zijn.

Beschouw de volgende specificatie:

Naam:	Voorbeeld extensie van Boolean		
Soorten:	X		
Operatoren:	leeg:		$\rightarrow X$
	erbij:	X	$\rightarrow X$
	is_leeg:	X	$\rightarrow$ Bool
Axioma's:	is_leeg(leeg)		= true
	is_leeg(erbij(s))		= false

Omdat geen enkele term van soort X equivalent blijkt met een andere term van soort X gaat de methode van de initiële algebra ervan uit dat alle termen van soort X verschillende waarden representeren. De term *erbij(leeg)* is daarom een waarde die verschilt van *erbij(erbij(leeg))*.

Een ander uitgangspunt gebaseerd op de zogenaamde zichtbaarheidsconcepten, voorgesteld door Giarratana et al. (1976) en verder ontwikkeld door Wand (1979), Hornung en Raulefs (1980) en Kamin (1980). Het centrale idee is dat termen *abstract identiek* zijn als ze door de operaties van het gegevenstype niet van elkaar kunnen worden onderscheiden. In bovenstaand voorbeeld zijn de termen *erbij(leeg)* en *leeg* te onderscheiden met behulp van de operatie *is_ leeg*. In het eerste geval levert *is_leeg* false als resultaat, in het tweede geval true. Maar de termen *erbij(leeg)* en *erbij(erbij(leeg))* zijn niet te onderscheiden, omdat er geen operatie of rij operaties is waarvan de uitvoer ons het verschil laat zien. Merk op de het zichtbaarheidsconcept gebruik moet maken van eerder gedefinieerde gegevenstypen waarvan we de waarden kennen. Het type Boolean voldoet hieraan en we nemen aan dat true onderscheidbaar is van false. Twee termen x en y van de soort s zijn dus abstract identiek dan en slechts dan als er geen Booleaanse term $t(v)$ met een enkele variabele v van soort s bestaat zodanig dat

$$t\,(x) = \text{true}$$
$$t\,(y) = \text{false}$$

In de overige gevallen worden de twee termen beschouwd als abstract verschillend. Terwijl de benadering met de initiële algebra ervan uitgaat dat alle termen waarvan niet kan worden bewezen dat ze equivalent zijn, daarom verschillend zijn, gaat de aanpak met een eindalgebra er alleen van uit dat abstract verschillende termen inderdaad verschillend zijn. Het verschil zit 'm in de behandeling van abstract identieke termen die niet equivalent zijn (bijvoorbeeld *erbij(leeg)* en *erbij(erbij(leeg))*). De methode met een initiële algebra gaat ervan uit dat dit verschillende waarden zijn. Bij een eindalgebra wordt ervan uitgegaan dat de

waarden gelijk zijn. Voor vele specificaties van gegevenstypen zijn de begin- en de eindalgebra gelijk. In bovenstaand voorbeeld zijn ze gelijk als we als axioma toevoegen:

erbij(erbij(s)) = erbij(s)

Een algebra A is een *eindalgebra* in een klasse van algebra's als er van elke algebra in de klasse een homomorfisme naar A bestaat. De eindalgebra die door de eindalgebramethode wordt gespecificeerd is de eindalgebra in de klasse van algebra's die voldoen aan de axioma's en tegelijk alle oude gegevenstypen beschermen (dat wil zeggen de eerder bestaande algebra's onverlet laten).

Een compromis, of liever een ruimere interpretatie van algebraïsche specificaties, is het in aanmerking laten komen van elke algebra die tussen de eind- en beginalgebra in ligt. Deze aanpak is equivalent met veronderstellen dat er geen aannamen worden gemaakt over abstract identieke termen die het gegevenstype representeren. Onder zo'n interpretatie stelt het gegevenstype niet langer een isomorfismeklasse voor.

Opgaven

1. Toon aan dat de verzameling C = {0, S0, SS0, ...} een canonieke vorm is voor axiomaverzameling R_8.

2. Toon aan dat de verzameling C = {0, S0, SS0, ...} een canonieke vorm is voor axiomaverzameling R_7.

3. Toon aan dat de volgende specificaties geen extensies zijn door aan te tonen dat ze niet EenvoudigeBoolean-volledig zijn of niet EenvoudigeBoolean-consistent of beide.

   Naam: Opg1 verrijking van EenvoudigeBoolean
   Operatoren:
   →: Bool × Bool → Bool
   Axioma's:
   true → false = false
   x → true = true

Naam: Opg2 verrijking van EenvoudigeBoolean
Operatoren:
$\rightarrow$: Bool × Bool $\rightarrow$ Bool
Axioma's:
$x \rightarrow x$ = true
true $\rightarrow x$ = false
false $\rightarrow x$ = true

Naam: Opg3 verrijking van EenvoudigeBoolean
Operatoren:
$\rightarrow$: Bool × Bool $\rightarrow$ Bool
Axioma's:
$x \rightarrow x$ = true
true $\rightarrow x$ = false

4. Bewijs dat een gegevenstype met een signatuur met één constante en één unaire operator kan worden uitgedrukt met één enkelvoudig axioma.

5. Geef een algebraïsche specificatie voor sequence(Z) met een unaire operator seq(Z) die een sequence van één element creëert die uit *z* bestaat, en '+' als concatenatie voor sequences, en *lengte(s)* die de lengte van de sequence *s* aflevert.

6. Geef een verrijking van de specificatie uit de vorige opgave met de operatie vervang(*a*,*b*,*c*), die sequence *a* aflevert met daarin het eerste voorkomen van de sequence *b* vervangen door de sequence *c*. Als de sequence *b* niet in sequence *a* voorkomt, levert de operator de sequence *a* als resultaat.

7. In het vorige hoofdstuk zijn zes Σ-algebra's gegeven. Bepaal de axiomaverzamelingen die de laatste vijf van die algebra's specificeren.

8. Geen een algebraïsche specificatie van de rationale getallen met de operaties 0, 1, +, -, *, / en een constante *E* die de fout 'delen door nul' aangeeft. Als *E* als operand bij een operator optreedt, wordt de waarde *E* als resultaat afgeleverd. Bedenk en beschrijf een canonieke vorm voor dit type.

9. Geef een algebraïsche specificatie van stack(Z) met foutenbehandeling. Creëer allereerst een nieuw type FouteBool met drie waarden: true, false en Boolfout. Maak vervolgens *Z*-fout een constante van het type *Z*. Laat tenslotte alle operaties op een foutwaarde een foutwaarde van de juiste soort afleveren.

10. Beschouw de algebraïsche specificatie van arrays (een voorbeeld uit Wand, 1979).

    Naam: IntArrays uitbreiding van Integer
    Soorten: A
    Operatoren:
    leeg: $\rightarrow$ A
    waarde: A $\times$ Int $\rightarrow$ Int
    wijzig: A $\times$ Int $\times$ Int $\rightarrow$ A
    Axioma's:
    waarde(leeg, x) = 0
    waarde(wijzig(a, x, y), z) = if ($x = z$)
    then y
    else waarde(a, z)

    Geef enkele voorbeelden van abstract identieke termen die niet equivalent zijn. Is voor dit gegevenstype een begin- of eindalgebra het meest geschikt? Welke axioma's zijn er nodig om de begin- en de eindalgebra gelijk te maken? (Merk op dat het toevoegen van zulke axioma's wel de beginalgebra verandert, maar niet de eindalgebra.)

11. Toon aan dat alle abstract identieke termen in de algebraïsche specificatie van verzamelingen ook equivalent zijn (waardoor de begin- en de eindalgebra gelijk zijn). Zij Nverzameling gelijk aan de algebraïsche specificatie van verzamelingen, maar zonder het eerste axioma. Toon aan dat de beginen de eindalgebra van Nverzameling verschillend zijn. Toon aan dat de eindalgebra gespecificeerd door Nverzameling isomorf is met de initiële algebra die door 'verzameling' wordt gespecificeerd.

Literatuur

De algebraïsche stof in dit hoofdstuk is grotendeels afgeleid uit het werk van de ADJ-groep, in het bijzonder van Goguen et al. (1978). Zilles (1974), Goguen et al. (1975) en Gutta (1975) hebben onafhankelijk van elkaar algebraïsche specificaties ontwikkeld. Tot het vroegere werk aan de axiomatische benadering van gegevenstypen behoort Hoare (1972a) en Standish (1973). Kamin (1979) doet een poging de terminologie te standaardiseren.

Er zijn vele semantische formalismen voor talen; een fraai overzicht is Pagan (1981). Zoals een gegevenstype als een eenvoudige programmeertaal kan worden beschouwd, zo kan een programmeertaal als een ingewikkeld gegevenstype worden beschouwd. Dat brengt ons tot het gebruik van algebraïsche specificaties voor het definiëren van programmeertalen (Wand, 1980; Broy en Wirsing, 1980; Cleaveland, 1980).

Er zijn vele vruchtbare onderzoeksterreinen en toepassingen op het gebied van algebraïsche specificaties, waaronder executeerbare specificaties zoals Affirm (Musser, 1979) en OBJ (Goguen en Tardo, 1979; Goguen, 1984) en equationeel programmeren (Hoffmann en O'Donnell, 1982, 1984). Volledigheid wordt besproken door Thiel (1984) en Jouannaud en Kirchner (1984).

Voor meer details over de lambdacalculus en modellen raadplege men Stoy (1977) voor een inleiding en Wadsworth (1976) of Barendregt (1981) voor alle details.

Bibliografie

Addyman, A.M. (1980), 'A Draft Proposal for Pascal,' *SIGPLAN Notices* **15**, 4, pp. 1-66.

Albano, A. (1983), 'Type Hierarchies and Semantic Data Models, *'Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, SIGPLAN Notices* **18**, 6, pp. 178-186.

Allen, J. (1978), *Anatomy of LISP*, McGraw-Hill, New York.

Anderson, E.R., F.C.Belz, en E.K.Blum (1976), 'SEMANOL(73): A Metalanguage for Programming the Semantics of Programming Languages,' *Acta Informatica* **6**, pp. 109-131.

Backus, John (1978a), 'The History of FORTRAN I, II, and III,' in *ACM SIGPLAN History of Programming Languages Conference,* June 1978, *SIGPLAN Notices* **13**, 8, pp. 165-180.

Backus, John (1978b), 'Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs,' *Communications of the ACM* **21**, 8, pp. 613-641.

Baker, T.P. (1982), 'A One-Pass Algorithm for Overload Resolution in Ada,' *ACM Transactions on Programming Languages and Systems* **4**, 4, pp.601-614.

Barendregt, H.P. (1981), *The Lambda Calculus: Its Syntax and Semantics,* North-Holland, Amsterdam.

Beech, D. (1970), 'A Structural View of PL/I,' *Computing Surveys* **2**, 1, pp. 33-64.

Berry, D.M., L.M. Chirica, J.B. Johnston, D.F. Martin, en A. Sorkin (1978), 'Time Required for Reference Count Management in Retention Block Structured Languages,' *Int. J. Comput. Inf. Sci.* **7**,, 1, pp. 11-64 (part 1); 7, 2, pp. 91-119 (part 2).

Berry, D.M., en R.L. Schwartz (1979), 'Type Equivalence in Strongly Typed Languages: One More Look,' *SIGPLAN Notices* **14**, 9, pp. 35-41.

Berry, D.M., en A. Sorkin (1978), 'Time Required for Garbage Collection in Retention Block-Structured Languages,' *Int. J. Comput. Inf. Sci.* **7**, 4, pp. 361-404.

Bert, D. (1983), 'Refinements of Generic Specifications with Algebraic Tools,' *Information Processing 83* pp. 815-820.

Blum, E.K., en F. Parisi-Presicce (1983), 'Implementation of Data Types by Algebraic Methods,' *Journal of Computer and System Sciences* **27**, pp. 304-330.

Borning, A.H., en D.H.H. Ingalls (1982), 'A Type Declaration and Inference System for Smalltalk,' *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages,* pp. 133-141.

Brainerd, W. (editor) (1978), 'Fortran 77,' *Communications of the ACM* **21**, 10, pp. 806-820.

Broy, M., en M. Wirsing (1980), 'Programming Languages as Abstract Data Types,' in M. Dauchet (ed.), Les Arbres en algèbre et en programmation, 5-ème Colloque de Lille, pp. 160-177, *Acta Informatica* **18**, pp. 47-64.

Broy, M., en M. Wirsing (1982), 'Partial Abstract Types,' *Acta Informatica* **18**, pp. 47-64.

Bruce, K.B., en A. Meyer (1984), 'The Semantics of Second Order Polymorphic Lambda Calculus,' in *Semantics of Data Types,* ed. G. Kahn, D.B. MacQueen, and G. Plotkin, *Lecture Notes in Computer Science* **173,** Springer-Verlag, pp. 131-144.

Burge, W.H. (1975), *Recursive Programming Techniques,* Addison-Wesley, Reading, Mass.

Burstall, R.M., en J.A. Goguen (1977), 'Putting Theories Together to Make Specifications,' *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* August 1977, Cambridge, Mass., pp. 1045-1058.

Burstall, R.M., D.B. MacQueen, en D.T. Sannella (1980), 'HOPE: An Experimental Applicative Language,' University of Edinburgh, Internal Report CSR-62-80, May 1980.

Burton, F.W., en B.J. Lings (1981), 'Abstract Data Types, Subtypes and Data Independence,' *The Computer Journal* **24**, 4, pp. 308-311.

Cardelli, L. (1984), 'A Semantics of Multiple Inheritance,' in *Semantics of Data Types,* edited by G. Kahn, D.B. MacQueen, and G. Plotkin, *Lecture Notes in Computer Science* **173**, Springer-Verlag, pp. 51-68.

Cleaveland, J.C., (1975), 'Meaning and Syntactic Redundancy,' in *New Directions in Algorithmic Languages,* Inst. de Recherche d'Informatique et d'Automatique, Rocquencourt, 1975, pp. 115-124.

Cleaveland, J.C., (1980), 'Programming Languages Considered as Abstract Data Types,' *Proceedings ACM 80,* Nashville, Tenn., October 1980.

Cohen, J. (1981), 'Garbage Collection of Linked Data Structures,' *Computing Surveys* **13**, 3, pp. 341-367.

Cohen, J., en A. Nicolau (1983), 'Comparison of Compacting Algorithms for Garbage Collection,' *ACM Transactions on Programming Languages and Systems* 5, 4, pp.532-553.

Coppo, M. (1983), 'On the Semantics of Polymorphism,' *Acta Informatica* **20**, 2, pp. 159-170.

Cormack, G.V. (1983), 'Extensions to Static Scoping,' *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems,* in *SIGPLAN Notices* **18**, 6, pp. 187-191.

Cousot, P., en R. Cousot (1977), 'Static Determination of Dynamic Properties of Generalized Type Unions,' *Proceedings of an ACM Conference on Language Design for Reliable Software,* in *SIGPLAN Notices* **12**, 3.

Curry, H.B., en R. Feys (1958), *Combinatory Logic I,* North-Holland, Amsterdam.

Dahl, O.J., en C.A.R. Hoare (1972), 'Hierarchical Program Structures,' in Dahl et al. (1972).

Dahl, O.J., C.A.R. Hoare, en E.W. Dijkstra (1972), *Structured Programming,* Academic Press, New York.

Dahl, O.J., B. Myhrhaug, en K. Nygaard (1968), *The Simula 67 Common Base Language,* Norwegian Computing Centre, Forskningsveien 1B, Oslo 3.

Damas, L., en R. Milner (1982), 'Principal Type-Schemes for Functional Programs.' *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages,* pp. 207-212.

Demers, A., J. Donahue, en G. Skinner (1978), 'Data Types as Values: Polymorphism, Type-checking, Encapsulation,' *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,* pp. 23-30.

Demers, A.J., en J.E. Donahue (1980a), 'Data Types, Parameters and Type Checking,' *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages,* pp. 12-23.

Demers, A.J., en J.E. Donahue (1980b), 'Type-Completeness' as a Language Principle' *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages,* pp. 234-244.

Dewar, R.B.K., A. Grand, Ssu-Cheng Liu, J.T. Schwartz, en E. Schonberg(1979), 'Programming by Refinement, as Exemplified by the SETL Representation Sublanguage,' *ACM Transactions on Programming Languages and Systems* **1**, 1, pp. 27-49.

Donahue, J.E. (1976), *Complementary Definitions of Programming Language Semantics, Lecture Notes in Computer Science* **42,** Springer-Verlag.

Duncan, D.M., (1979), 'Bibliography on Data Types,' *SIGPLAN Notices* **14**, 11, pp. 31-59.

Eggert, P.R. (1981), 'Detecting Software Errors before Execution,' UCLA Computer Science Dept. Report No. CSD-810402, April 1981.

Falkoff, A.D., en K.E.Iverson (1973), 'The Design of APL,' *IBM Journal of Research and Development* July 1973, pp. 324-334; ook in Horowitz (1983), pp. 212-222.

Falkoff, A.D., en K.E. Iverson (1978), 'The Evolution of APL.' in Wezelblat (1978).

Fleck, A.C. (1978), 'Formal Models for String Patterns,' in Yeh (1978), pp. 216-240.

Friedman, D.P., en D.S. Wise (1976), 'CONS Should Not Evaluate Its Arguments,' in *Automata, Languages and Programming,* S. Michaelson, en R. Milner (editors), Edinburgh University Press, Edinburgh.

Gannon, J.D. (1977), 'An Expirimental Evaluation of Data Type Conventions,' *Communications of the ACM* **20**, 8, pp. 584-595.

Gannon, J.D., en J.J. Horning (1975), 'Language Design for Programming Reliability,' *IEEE Transactions on Software Engineering* **1**, 2, June, 1975.

Ganzinger, H. (1983), 'Parameterized Specifications: Parameter Passing and Implementation with Respect to Observability,' *ACM Transactions on Programming Languages and Systems* **5**, 3, pp. 318-354.

Gehani, N,. en D. Gries (1977), 'Some Ideas on Data Types in High Level Languages', *Communications of the ACM* **20**, pp. 414-420.

Gerhart, S.L., en L. Yelowitz (1976), 'Observations of Fallibility in Applications of Modern Programming Methodologies,' *IEEE Transactions on Software Engineering*, **2**, 3, pp. 195-207.

Geschke, C.M., J.H. Morris Jr. en E.H. Satterwaite (1977), Early Experience with Mesa,' *Communications of the ACM* **20**, 8, pp. 540-553.

Giarratana, V., F. Gimona, en U. Montanari (1976), 'Observability Concepts in Abstract Data Type Specifications, in A. Mazurkiewicz (editor), *Lecture Notes in Computer Science* **45**, Springer-Verlag, pp. 576-587.

Goguen, J.A. (1984), 'Parameterized Programming,' *IEEE Transactions on Software Engineering* **10**, 5, pp. 528-543.

Goguen, J.A., en J.J. Tardo (1979), 'An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications,' *Proceedings IEEE Specifications of Reliable Software.*

Goguen, J.A., en J.W. Thatcher (1974), 'Initial Algebra Semantics,' *Proceedings of the 15th IEEE Symposium on Switching and Automata Theory,* New Orleans, October 1974.

Goguen, J.A., en J.W. Thatcher, en E.G. Wagner (1978), An Initial Algebra

Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in Yeh (1978), pp. 80-149.

Goguen, J.A., J.W. Thatcher, E.G. Wagner, en J.B. Wright (1975), 'Abstract Data Types as Initial Algebras and the Correctness of Data Representations,' *Proceedings, Conference on Computer Graphics, Pattern Recognition, and Data Structures,* pp. 89-93.

Goldberg, A., en D. Robson (1983), *Smalltalk-80, The Language and Its Implementation,* Addison-Wesley, New York.

Goodwin, J.W. (1981), 'Why Programming Environments Need Dynamic Data Types, '*IEEE Transactions on Software Engineering* **7**, 5, Sept. 1981.

Gordon, M.J., A.J.Milner, en C.P. Wadsworth (1979), *Edinburgh LCF, Lecture Notes in Computer Science* **78**, Springer-Verlag.

Gries, D. (editor) (1978), *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, Springer-Verlag, New York.

Griswold, R.E. (1982), ' The Evaluation of Expressions in Icon,' *ACM Transactions on Programming Languages and Systems* **4**, 4, pp. 563-584.

Griswold, R.E. (1983), *ICON Programming Language*, Prentice-Hall, Englewood Cliffs, N.J.

Griswold, R.E., J.F. Poage, en I.P.Polansky (1971), *The SNOBOL 4 Programming Language* (Second Edition), Prentice-Hall, Englewood Cliffs, N.J.

Gull, W.E., en M.A. Jenkins (1979), 'Decisions for 'Type' in APL,' *Conference Record of the Sixth Annual Symposium on Principles of Programming Languages*, pp. 190-196.

Guttag, J.V. (1975), 'The Specifications and Application to Programming of Abstract Data Types,' Ph.D. Thesis, University of Toronto, Department of Computer Science, CSRG-59.

Guttag, J.V. (1977), 'Abstract Data Types and the Devolopment of Data Structures.' *Communications of the ACM* **20**, 6, pp. 396-404.

Guttag, J.V., en J.J. Horning (1983), 'An Introduction to the Larch Shared Languages,' *Information Processing* **83**, pp. 809-814.

Guttag, J.V., E. Horowitz, en D.R. Musser (1978), ' The Design of Data Type Specifications,' in Yeh (1978), pp.61-80.

Habermann A.N., en D.E. Perry (1983), *Ada for Experienced Programmers,* Addison Wesley, Reading, Mass.

Harland, D.M. (1984), *Polymorphic Programming Languages: Design and Implementation,* Ellis Horwood, Chichester, England.

Harland, D.M., en H.I.E. Gunn (1982), 'Another Look at Enumerated Types,' SIGPLAN Notices **17**, 7, pp. 62-71.

Harle, J. (1983), 'The Proposed Standard for BASIC,' *SIGPLAN Notices* **18**, 5, pp. 25-40.

Henderson, P. (1980), *Functional Programming: Application and Implementation*, Prentice-Hall, Englewood Cliffs, N.J.

Henderson, P., en J.H. Morris Jr. (1976), 'A Lazy Evaluator,' *Third ACM Symposium on Principles of Programming Languages*, pp. 95-103.

Hindley, R. (1969), 'The Principal Type-Scheme of an Object in Combinatory Logic,' *Transactions of the American Mathematical Society* **146**, pp. 29-60.

Hoare, C.A.R. (1972a), 'Notes on Data Structuring,' in Dahl et al. (1972).

Hoare, C.A.R. (1972b), 'Proof of Correctness of Data Representations,' *Acta Informatica* **1**, pp. 271-281; also in Gries (1978).

Hoare, C.A.R., en N. Wirth (1973), 'An Axiomatic Definition of the Programming Language PASCAL,' *Acta Informatica* **2**, pp. 335-355.

Hoffmann, C.M., en M.J.O.'Donnell (1982), 'Programming with Equations,' *ACM Transactions on Programming Languages and Systems* **4**, 1, pp.83-112.

Hoffmann, C.M., en M.J.O'Donnell (1984), 'Implementation of an interpreter for abstract equations,' *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 11-120.

Hornung, G., en P. Raulefs (1980), 'Terminal Algebra Semantics and Retractions for Abstract Data Types,' in J. DeBakker, en J. Leeuwen (editors), *Lecture Notes in Computer Science* **85**, Springer-Verlag, pp. 310-323.

Horowitz, E. (editor) (1983), *Programming Languages: A Grand Tour*, Computer Science Press, Rockville, Maryland.

Ichbiah, J.D., J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, B.A. Wichmann (1979), 'Rationale for the design of the Ada programming language,' *SIGPLAN Notices* **14**, 6, part B.

Jackson, M.A. (1977), 'COBOL,' in *Software Engineering*, ed. R.H.Perrott, pp. 47-57, Academic Press, New York.

Jensen, K., en N. Wirth (1974), *PASCAL User Manual and Report* (second Edition), Springer-verlag, New York en Berlin.

Jouannaud, J.P., en H. Kirchner (1984), 'Completion of a Set of Rules Modulo a Set of Equations,' *Conference Record of the Eleventh Annual ACM Symposium of Principles of Programming Languages*, pp. 83-92.

Kamin, S. (1979), 'Some Definitions for Algebraic Data Type Specifications,' *SIGPLAN Notices* **14**, 3.

Kamin, S. (1980), 'Final Data Specifications: a New Data Type Specification Method,' *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 131-138.

Kamin, S.(1983), 'Final Data Types and Their Speicifaction,' *ACM Transactions on Programming Languages and Systems* **5**, 1, pp. 97-123.

Kamin, S., en M. Archer (1984), 'Partial Implementations of Abstract Data Ty-

pes: A Dissenting view on Errors,' in *Semantics of Data Types*, ed. G. Kahn, D.B. MacQueen, en. Plotkin, *Lecture Notes in Computer Science* **173**, Springer-Verlag, pp. 317-336.

Kaplan, M.A., en J.D. Ullman (1980), 'A Scheme for the Automatic Inference of Variable Types,' *Journal of the ACM* **27**, 1.

Karr, M., en D.B. Loveman III (1978), 'Incorporation of Units into Programming Languages,' *Communications of the ACM* **21**, 5, pp. 385-391.

Kernighan, B.W., en D.M. Ritchie (1978), *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J.

Knuth, D.E. (1968), 'Semantics of Context-Free Languages,' *Mathematical Systems Theory* **2**, pp. 127-145.

Knuth, D.E. (1973), *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

Lampson, B.W., J.J. Horning, R.L. London, J.G. Mitchell, en G.J. Popek (1977), 'Report on the Programming Language Euclid,' *SIGPLAN Notices* **12**, 2, pp. ii-79.

Leivant, D. (1983a), 'Polymorphic type inference.' *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages.* pp. 88-98.

Leivant, D, (1983b), 'Structural semantics for polymorphic data types,' *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*. pp. 155-166.

Liskov, B., A. Snyder, R. Atkinson, en C. Schaffert (1977), 'Abstraction Mechanisms in CLU,' *Communications of the ACM* **20**, 8, pp. 564-576.

Lucas, P., P. Lauer, en H. Stigleitner (1968), ' Method and Notation for the Formal Definition of Programming Languages,' Technical Report 25.087, IBM Laboratory, Vienna.

MacLennan, B.J. (1982), 'Values and Objects in Programming Languages,' *SIGPLAN Notices* **17**, 12, pp. 70-79.

MacQueen, D.B., G. Plotkin, en R. Sethi (1984), 'An Ideal Model for Recursive Polymorphic Types,' *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 165-174.

MacQueen, D.B., en R. Sethi (1982), 'A Semantic Model for the Types of Applicative Languages,' *Proceedings of 1982 ACM Symposium on LISP and Functional Programming*, pp. 243-252.

Majster, M.E. (1977), 'Limits of the 'Algebraic' Specification of Abstract Data Types,' *SIGPLAN Notices* **12**, 10, pp. 37-42.

Majster, M.E. (1979), 'Treatment of Partial Operations in the Algebraic Specification Technique,' *Proceedings of the Specifications of Reliable Software Conference*, April 1979, pp. 190-197.

Marcotty, M., H.F. Ledgard, en G.V. Bochmann, 'A Sampler of Formal Definitions,' *Computing Surveys* **8**, 2, pp. 191-276.

McCarthy, J. (1960) 'Recursive Functions of Symbolic Expressions and Their Computation by Machine, part 1,' *Communications of the ACM* **3**, 4, pp. 184-195.

McCarthy, J., en M. Levin (1965), *LISP 1.5 Programmers Manual*, MIT Press, Cambridge, Mass.

McCracken, D.D. (1975), *Digital Computer Programming*, John Wiley & Sons, Inc., New York.

McCracken, N. (1984), 'The Typechecking of Programs with Implicit Type Structure,' in *Semantics of Data Types*, ed. G. Kahn, D.B. MacQueen, and G. Plotkin, *Lecture Notes in Computer Science* **173**, Springer-Verlag, pp. 301-316.

Meertens, L.G.L.T. (1983), 'Incremental Polymorphic Type Checking in 'B', ' *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pp. 265-275.

Meyer, A.R. (1982), 'What Is A Model of the Lambda Calculus?,' *Information and Control* **52**, 1, pp. 87-122.

Miller, T.C. (1979)), 'Type Checking in an Imperfect World,' *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 237-243.

Milner, R. (1978), 'A Theory of Type Polymorphism in Programming,' *Journal of Computer and System Science* **17**, pp. 348-375.

Mitchell, J.C. (1984), 'Coercion and Type Inference,' *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 175-185.

Mitchell, J.G., en B. Wegbreit (1978), 'Schemes: A High-level Data Structuring Concept,' in Yeh (1978), pp. 150-184.

Moffat, D.V. (1981), 'Enumerations in Pascal, Ada, and Beyond,' *SIGPLAN Notices* **16**, 2, pp. 77-82.

Morris, J.H. (1973). 'Types Are Not Sets,' *Conference Record of ACM Symposium on the Principles of Programming Languages*, pp. 120-124, October 1973.

Musser, D.R. (1979), 'Abstract Data Type Specification in the AFFIRM System,' *Proceedings of the Specifications of Reliable Software Conference*, April 1979.

Naur, P. (editor) (1963), 'Revised Report on the Algorithmic Language ALGOL 60,' *Communications of the ACM* **6**, 1, pp. 1-20.

Nicholls, J.E. (1975), *The Structure and Design of Programming Languages*, Addison-Wesley, Reading, Mass.

Oyamaguchi, M. (1985), 'On the Data Type Extension Problem for Algebraic Specifications,' *Theoretical Computer Science* **35**, pp. 329-336.

Pagan, F.G. (1981), *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Englewood Cliffs, N.J.

Popek, G.J., J.J. Horning, B.W. Lampson, J.G. Mitchell, en R.L. London (1977), 'Notes on the Design of Euclid,' *SIGPLAN Notices* **12**, 3, pp. 11-19; also in Horowitz (1983), pp. 252-260.

Reynolds, J.C. (1970), 'GEDANKEN - A Simple, Typeless Language Based on the Principle of Completeness and on the Reference Concept,' *Communications of the ACM* **13**, 5, pp. 308-319.

Reynolds, J.C. (1975), 'User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction' in *New Directions in Algorithmic Languages,* Inst. de Recherche d'Informatique et d'Automatique, Rocquencourt, 1975, pp. 157-168; also in Gries (1978).

Reynolds, J.C. (1978), 'Syntactic Control of Interference,' *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 39-46.

Reynolds, J.C. (1983), 'Types, Abstractions and Parametric Polymorphism,' *Information Processing* **83**, pp. 513-523.

Reynolds, J.C. (1984), 'Polymorphism Is Not Set-Theoretic,' in *Semantics of Data Types*, edited by G. Kahn, D.B. MacQueen, and G. Plotkin, *Lecture Notes in Computer Science* **173**, Springer-Verlag, pp. 145-156.

Richards, M. (1969), 'BCPL: A Tool for Compiler Writing and System Programming,' *Proceedings of AFIPS Spring Joint Computer Conference* **34**, pp. 557-566.

Robinson, J.A. (1965), 'A Machine-Oriented Logic Based on the Resolution Principle,' *Journal of the ACM* **12**, 1, pp. 23-41.

Schonberg, E., J. Schwartz, en M. Sharir (1981), 'An Automatic Technique for Selection of Data Representation in SETL Programs,' *ACM Transactions on Programming Languages and Systems* **3**, 2.

Scott, D.S. (1970), 'Lattice Theory, Data Types, and Semantics,' in *Formal Semantics of Programming Languages*, edited by R. Rustin, 2nd Courant Computer Science Symposium, Prentice-Hall (1972).

Scott, D.S. (1976), 'Data types as lattices,' *SIAM J. on Computing* **5**, 3, pp. 522-586.

Shaw, M. (1976), 'Research Directions in Abstract Data Types,' *Conference Proceedings on Data: Abstraction, Definition and Structure, SIGPLAN Notices* **8**, 2, pp. 66-68.

Sherman, M. (1984), 'Paragon: Novel Uses of Type Hierarchies for Data Abstraction,' *Conference Record of the Eleventh Annual ACM Symposium on*

Principles of Programming Languages, pp. 208-217.

Standish, T.A. (1973), 'Data Structures: An Axiomatic Approach,' BBN Report 2639, Bolt Beranek and Newmann, Cambridge, Mass.

Standish, T.A. (1980), *Data Structures Techniques*, Addison-Wesley, Reading, Mass.

Stoy, J.E. (1977), Denotational Semantics: *The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass.

Stroustrup, B. (1986), *The C ++ Programming Language*, Addison-Wesley, Reading, Mass.

Suzuki, N. (1981), 'Inferring Types in Smalltalk,' *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, ACM.

Taghva, K. (1983), 'Constructive Fully Abstract Models of Typed Lambda-Calculi,' CSR 159, Computer Science Dept., New Mexico Tech, Socorro, N.M., December 1983.

Tanenbaum, A.S. (1976), 'A Tutorial on Algol 68' *Computing Surveys* **8**, 2, pp. 155-190; also in Horowitz (1983), pp. 89-124.

Tenenbaum, A.M. (1974), 'Compile Time Type Determination in SETL,' *Proceedings of the ACM 1974 Annual Conference*, November 1974.

Tennent, R.D. (1973), 'Mathematical Semantics of SNOBOL4,' *Conference Record of ACM Symposium on Principles of Programming Languages*, pp. 95-107.

Tennent, R.D. (1976), 'The Denotational Semantics of Programming Languages,' *Communciations of the ACM* **19**, 8, pp. 437-453.

Tennent, R. (1978), 'Another Look at Type Compatibility in Pascal.' *Software Practice and Experience* **8**, pp. 429-437.

Tennent, R.D. (1981), *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs, N.J.

Thatcher, J.W., E.G. Wagner, en J.B. Wright (1979), 'Data Type Specification: Parameterization and the Power of Specification Techniques,' IBM Research Report RC 7757, 34 pages; also in *ACM Transactions on Programming Languages and Systems* **4**, 4, pp. 711-732.

Thiel, J.J. (1984), 'Stop Losing Sleep over Incomplete Data Type Specifications,' *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 76-82.

van Wijngaarden, A., B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, en R.G. Fisker (ed.) (1975), 'Revised Report on the Algorithmic Language Algol 68,' *Acta Informatica* **5**, pp. 1-236.

Wadsworth, C.P. (1976), 'The Relation between Computational and Denotational Properties for Scotts D_∞–Models of the Lambda-Calculus,' SIAM Jour-

nal on Computing 5, pp. 488-521.

Wampler, S.B., en R.E. Griswold (1983), 'The Implementation of Generators and Goal-Directed Evaluation in Icon,' *Software Practice and Experience* **13**, pp. 495-518.

Wand, M. (1979), 'Final Algebra Semantics and Data Type Extensions,' *Journal of Computers and System Sciences* **19**, 1, pp. 27-44.

Wand, M. (1980), 'First-Order Identities as a Defining Language,' *Acta Informatica* **14**, pp. 337-357.

Wand, M. (1984), 'A Types-as-Sets Semantics for Milner-Style Polymorphism,' *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 158-164.

Wasserman, A.I. (editor) (1980), *Tutorial: Programming Language Design*, IEEE Catalog No. EHO 164-4.

Wegbreit, B. (1974), 'The Treatment of Data Types in EL1,' *Communications of the ACM* **17**, 5, pp. 251-264.

Wegner, P. (1972), 'The Vienna Definition Language,' *Computing Surveys* **4**, 1, pp. 5-63.

Welsh, J., W.J. Sneeringer, en C.A.R. Hoare (1977), 'Ambiguities and Insecurities in Pascal,' *Software Practice and Experience* **7**, pp. 685-696.

Wexelblat, R.L. (editor) (1978), *ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Notices* **13**, 8.

White, J.R. (1983), 'On the Multiple Implementation of Abstract Data Types Within a Computation,' *IEEE Transactions on Software Engineering* **9**, 4, pp. 395-410.

Wirsing, M., P. Pepper, H. Partsch, W. Dosch, en M. Broy (1983), 'On Hierarchies of Abstract Data Types,' *Acta Informatica* **20**, 1, pp. 1-33.

Wirth, N. (1971), 'The Programming Language Pascal,' *Acta Informatica*, **1**, 1, pp. 35-63.

Wirth, N. (1977), 'MODULA: a Language for Modular Multiprogramming,' *Software Practice and Experience*,' **7**, pp. 3-35; also in Horowitz (1983).

Wirth, N. (1980), *MODULA-2*, Berichte des Instituts fur Informatik No. 36, ETH, Zurich.

Wulf, W.A., R.L. London, en M. Shaw (1976), 'An Introduction to the Construction and Verification of Alphard Programs,' *IEEE Transactions on Software Engineering* **2**, 4, pp. 253-265.

Wulf, W.A., D.B. Russell, en A.N. Habermann (1971), 'BLISS; A Language for Systems Programming,' *Communications of the ACM* **14**, 12, pp. 780-790.

Yeh, R.T. (editor) (1978), *Current Trends in Programming Methodology, Vol. IV, Data Structuring*, Prentice-Hall, Englewood Cliffs, N.J.

Zilles, S.N. (1974), 'Algebraic Specification of Data Types,' Project MAC Pro-

gress Report 11, MIT, Cambridge, Mass., pp. 28-52.

Trefwoordenlijst

Dit is het eerste boek dat zich geheel richt op de gegevenstypen van programmeertalen. In *Een inleiding tot gegevenstypen* wordt een overzicht gegeven van het gebruik van gegevenstypen en worden ook vele verwante zaken diepgaand behandeld.
Gegevenstypen worden onderzocht aan de hand van de grote verscheidenheid van opvattingen waarvan wordt uitgegaan in vele verschillende programmeertalen, waaronder Ada, ALGOL 68, C, ML, Pascal en PL/I.
In het boek worden onder meer de volgende punten behandeld:

- Typecontrole
- Geheugenbeheer
- Abstracte gegevenstypen
- Polymorfisme
- Specificaties
- Algebraïsche specificaties

Een inleiding tot gegevenstypen helpt de lezer met een collectie opgaven en een literatuuroverzicht aan het eind van elk hoofdstuk. Het boek is zo opgezet dat het als studieboek en als naslagwerk kan worden gebruikt. Het veronderstelt kennis van programmeertalen.

> ACADEMIC SERVICE ADDISON-WESLEY

ISBN 90 6233 292 7
NUGI 852